

AD-A152 857

A TEST METHODOLOGY FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT(U) AIR FORCE INST OF  
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.

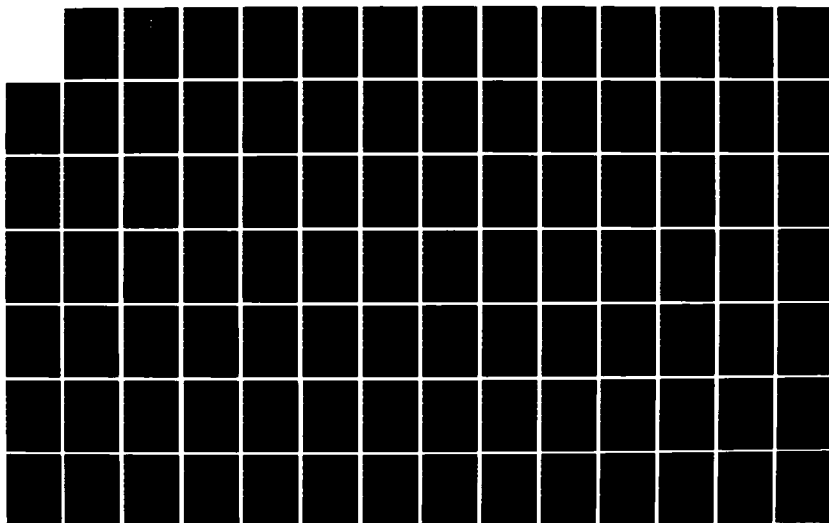
14

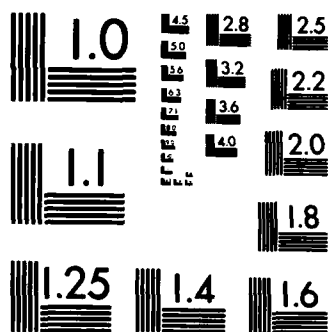
UNCLASSIFIED

K A SHOMPER DEC 84 AFIT/GCS/ENG/84D-26

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

DTIC

①

AD-A152 857



A TEST METHODOLOGY  
FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

Keith A. Shomper  
Second Lieutenant, USAF

AFIT/GCS/ENG/84D-26

DTIC FILE COPY

DEPARTMENT OF THE AIR FORCE  
AIR UNIVERSITY

**AIR FORCE INSTITUTE OF TECHNOLOGY**

DTIC  
ELECTRONIC  
APR 29 1985  
S E

Wright-Patterson Air Force Base, Ohio

This document is  
for public  
distribution  
REPRODUCED AT GOVERNMENT EXPENSE

8 R 04 86 004

AFIT/GCS/ENG/84D-26

**A TEST METHODOLOGY  
FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT**

**THESIS**

**Keith A. Shomper  
Second Lieutenant, USAF**

**AFIT/GCS/ENG/84D-26**

**Approved for public release; distribution unlimited**

**DTIC  
NOTE  
APR 29 1985  
S E D**



A TEST METHODOLOGY  
FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT

THESIS

Presented to the Faculty of the School of Engineering  
of the Air Force Institute of Technology  
Air University

In Partial Fulfillment of the  
Requirements for the Degree of  
Master of Science in Electrical Engineering



Keith A. Shomper, B.A.  
Second Lieutenant, USAF

December 1984

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
Type	
Date	
Code	
Other	
Date	
A-1	

Approved for public release; distribution unlimited

## Preface

The purpose of this investigation was to develop a methodology which would support software lifecycle testing in an automated and interactive development environment. This thesis and its associated program is the result. Currently, only a single test is implemented under this methodology; however, as the groundwork is now laid, my hope is that development on this methodology would continue to complete this much needed tool at AFIT.

These past eighteen months have not been easy, but they have been rewarding. I owe a number of people thanks for their support during this period. I would like to thank Dr. Gary Lamont for his guidance, suggestions, corrections, and support in creating this thesis; it undoubtedly made in much better. I would also like to thank Lt. Colonel Hal Carter for making me first feel welcome at the School. I thank my classmates, particularly Lt. Jim Kirkpatrick and Captain Paul Moore, for their example and encouragement in those final thesis days.

Most importantly, I want to thank my fiancée, Vickie, for the time she gave up and the encouragement she gave me so I could finish this work. I dedicate this thesis to my Mom and Dad, my family, Jack and Steve, and my friends at church, may our Lord Jesus Christ help them in all their work as he has helped me in mine.

Keith A. Shomper

## Table of Contents

	Page
Preface . . . . .	ii
List of Figures . . . . .	vi
List of Tables . . . . .	vii
Abstract . . . . .	viii
I. Introduction . . . . .	I-1
Thesis Objective . . . . .	I-1
Background . . . . .	I-1
Scope . . . . .	I-8
Assumptions . . . . .	I-9
Summary of Current Knowledge . . . . .	I-9
Standards . . . . .	I-9
Approach . . . . .	I-10
Materials and Equipment . . . . .	I-13
Summary . . . . .	I-13
II. Requirements Definition . . . . .	II-1
Introduction . . . . .	II-1
Testing Needs . . . . .	II-3
Error Types and Tests . . . . .	II-6
Requirements Considerations . . . . .	II-9
General Requirements . . . . .	II-12
Requirements Representations . . . . .	II-22
Functional Model . . . . .	II-24
Summary . . . . .	II-27
III. Preliminary Design . . . . .	III-1
Introduction . . . . .	III-1
Preliminary Design Concerns . . . . .	III-2
SDW Test Methodology Configuration Model . . . . .	III-7
Resolution of Requirements . . . . .	III-10
The Preliminary Design Representation . . . . .	III-17
The Preliminary Design . . . . .	III-19
Summary . . . . .	III-22

IV.	Detailed Design . . . . .	IV-1
	Introduction . . . . .	IV-1
	Detailed design Issues . . . . .	IV-2
	Methods for Creating a Detailed Design . . . . .	IV-13
	Presentation of the Detailed Design . . . . .	IV-25
	The Detailed Design . . . . .	IV-26
	Summary . . . . .	IV-28
V.	Implementation . . . . .	V-1
	Introduction . . . . .	V-1
	Creating a "Good" Implementation . . . . .	V-2
	The SDW Test Methodology Implementation . . . . .	V-6
	SDW Test Methodology Version 1.0 . . . . .	V-23
	Summary . . . . .	V-30
VI.	Recomendations and Conclusions . . . . .	VI-1
	Introduction . . . . .	VI-1
	Recommendations for Future Development . . . . .	VI-1
	Conclusions . . . . .	VI-3
	Summary . . . . .	VI-4
Appendix A:	Requirements Definition Model . . . . .	A-1
Appendix B:	Preliminary Design . . . . .	B-1
Appendix C:	Detailed Design . . . . .	C-1
Appendix D:	Data Dictionary . . . . .	D-1
Appendix E:	Configuration Model Justification . . . . .	E-1
Bibliography	. . . . .	BIB-1
Vita	. . . . .	V-1

## List of figures

Figure		Page
1.	The Traditional Software Lifecycle . . . .	I-4
2.	The Software Lifecycle and Integrated Testing . . . . .	I-4
3.	Organizational Relationships of QA and IV&V Teams . . . . .	II-4
4.	The Economics of Software Quality Assurance . . . . .	II-7
5.	SADT Symbols . . . . .	II-24
6.	Functional Model AO Diagram . . . . .	II-25
7.	SDW Configuration Model . . . . .	III-8
8.	Preliminary Design AO Diagram . . . . .	III-20
9.	Preliminary Design Node 1.1.3 . . . . .	III-21
10.	Meta Stepwise Refinement as an Iterative Process . . . . .	IV-24
11.	Conversion of the Logical Structure to a Physical Design Structure . . . . .	IV-27
12.	A Pipeline of Testing Sieves . . . . .	V-3
13.	The Language Selection Domains . . . . .	V-7
14.	Validation, a Means of Reflection . . . .	V-8
15.	Standard File Header . . . . .	V-25
16.	Standard Module Header . . . . .	V-25
17.	The SDW Test Methodology as a Modular System . . . . .	V-30

## List of Tables

Table		Page
I.	Structure Chart Graphical Symbols . . . .	III-18
II.	PAD Graphical Symbols . . . . .	IV-16
III.	Structure Chart Graphical Symbols . . . .	IV-17
IV.	ANSI Flowchart Graphical Symbols . . . . .	IV-19
V.	Common Pseudocode Characteristics . . . .	IV-21
VI.	PDL Control Constructs . . . . .	IV-22
VII.	FORTRAN Control Constructs . . . . .	V-12
VIII.	Pascal Control Constructs . . . . .	V-13

Abstract

The purpose of this investigation is to examine the concept of lifecycle testing, in particular, in the AFIT education/research environment. A result of this investigation is the SDW Test Methodology, an automated/interactive testing tool to aid the software engineer in lifecycle testing.

Five areas of testing are identified during this investigation which are common to software development in the analysis and design phases. These areas are consistency, correctness, clearness, completeness, and traceability. By providing automated and interactive functions to test for the errors associated with these areas, the AFIT student software engineer is relieved from the hand-testing techniques which heretofore have been employed preceeding implementation. The Test Methodology supports the latter half of lifecycle testing by providing interfaces for an assortment of static and dynamic analysis testing tools.

On a wider scale, the ideas and conclusions presented herein are applicable in all software development environments employing a structured approach to analysis,



design, and implementation. However, the software tests are not generally applicable to all environments and may require modifications. The SDW Test Methodology is initially hosted on a VMS VAX 11/780 and requires interfaces to the INGRES Relational Data Base Manager and the Data Dictionary Generation Tool (58).

## Introduction

### Thesis Objective

The objective of this thesis investigation is to analyze the software development lifecycle, at each phase of the lifecycle, to determine what criteria must be met to insure a complete testing methodology for the entire lifecycle. A preliminary design outlines how this methodology might be implemented on an existing software system, namely the Software Development Workbench (SDW) at the Air Force Institute of Technology's Digital Engineering Laboratory. Following the preliminary design, the focus of the thesis effort tightens as the detailed design is presented. An implementation of a particular section of the lifecycle testing methodology concludes this investigation.

### Background

"Software," "lifecycle," "software engineering," "bugs," and "program debuggers": these are terms that had no meaning or radically different meanings thirty years ago (4). Indeed, the field in which they are commonly used, computer science/computer engineering, did not exist. Today the professional software engineer is expected to create complicated algorithms and data bases according to sound engineering practices so that they may be created with confidence (56:368). These expectations, though burdensome,

exist and must be satisfied as the magnitude and the complexity of software systems continues to expand.

In the hustle and bustle of early programming, because computer memory was small, the programs that used these machines were correspondingly small. Consequently the program structures were relatively simple and no formal methodology for software development was needed. Most programmers of this era would simply code with little regard for documentation or design. Their approach was often unstructured, haphazard, random, or born out of a "stroke-of-genius". As computing power expanded, the new programs developed to exercise and control the full capabilities of the new machines grew in complexity.

This complexity could no longer be handled by the old random methods. The complexity wrought confusion, but it also illustrated the need for a new type of discipline to guide the production of software. This discipline has evolved into a set of theories and methodologies which is now known as software engineering.

Software engineering is built upon a foundation called the software lifecycle. It is the lifecycle that breaks the engineering task into manageable pieces. It also organizes the flow of production, giving order to the software product. Mark Smith et al. states, that "This division into phases is indispensable to the construction of a 'valid' or

'reliable' product" (56:368). The lifecycle approach is simply a twist to the idea that problems are best solved when divided. Divide and conquer has always been a good approach to attack a complex problem (27:5). Generals used this method by setting war objectives, creating battle plans, choosing the right men to implement the plans, attacking as planned, and finally, occupying the territory; they would never simply grab their guns and attack. Engineers use this method by contracting to build a structure, designing the structure to specific specifications, creating blueprints to show exact measurements and materials, and building the structure from the blueprints. During this time inspectors make certain that the building does not interfere with city planning, that it satisfies the zoning codes and city restrictions, that the materials meet defined standards, and that the internal structure, the wiring and plumbing, meet or exceed safety standards.

This same type of planning must exist in software engineering if quality reliable software is the objective (48:14). This phase planning in software engineering is called the software development lifecycle. The phases outlined in this thesis are the requirements definition phase, the preliminary design phase, the detailed design phase, the implementation or coding phase, the integration

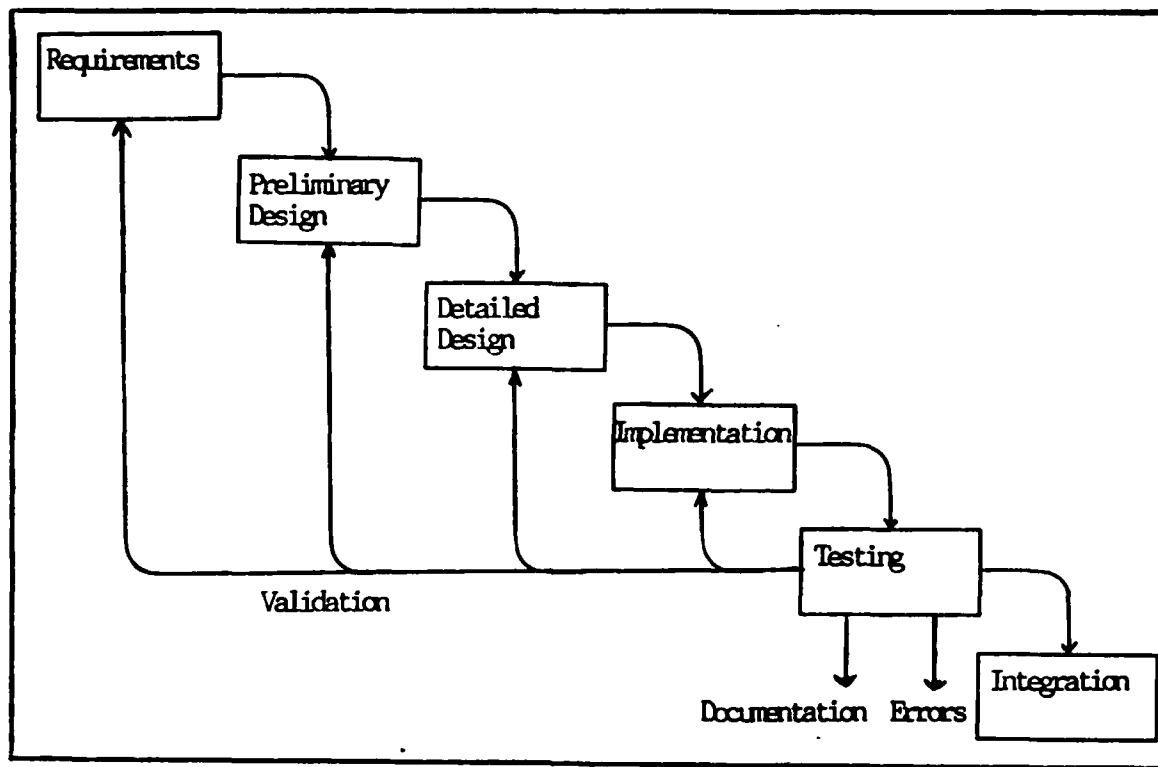


Figure 1: The Traditional Software Lifecycle

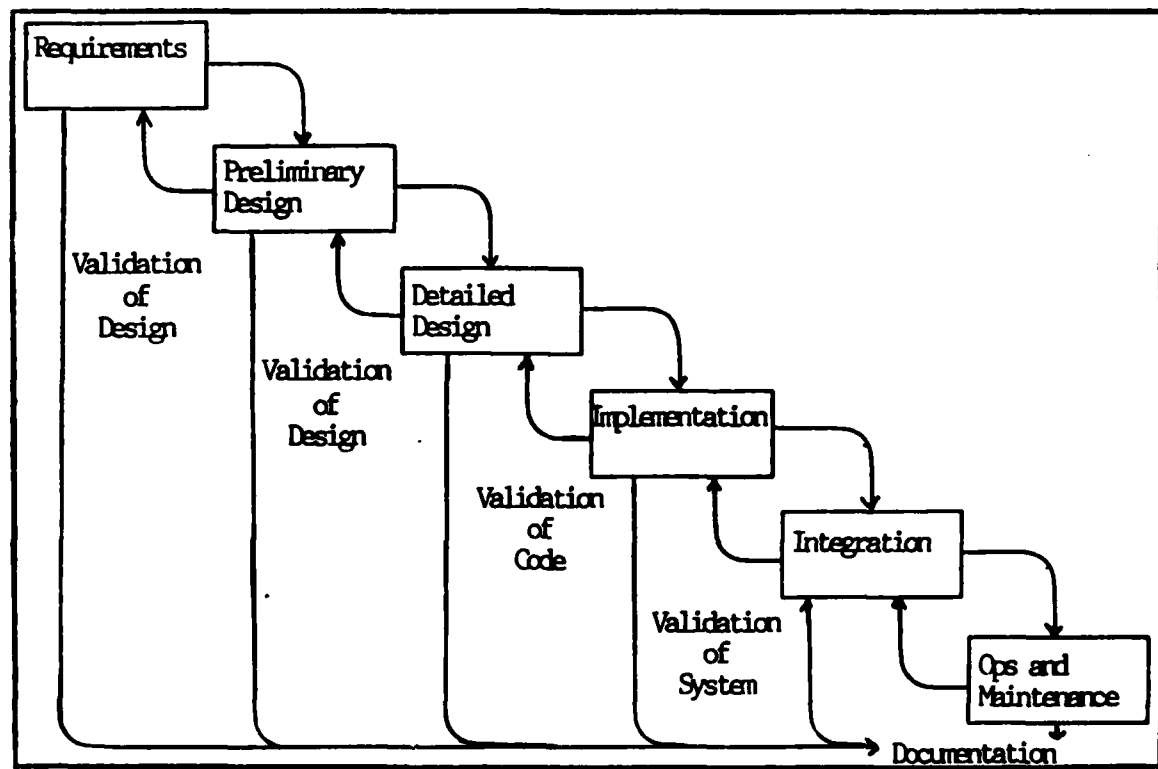


Figure 2: The Software Lifecycle and Integrated Testing

phase, and the operations and maintenance phase (figure 2). Similar to the roles of the building inspector and city planner, testing, or validation, should occur throughout the "construction" of the software. This is done to insure that the original objectives are satisfied, because that target may be lost or grow dim over the life of a project; particularly if it is delayed.

Traditionally the lifecycle included a testing phase which was performed sequentially between the coding and implementation phases (30:1), see figure 1. This type of isolated testing insured that the program ran sample data without errors. Once this testing was complete, the software was delivered to the user for integration and operation. But a fallacy existed because of the isolation of this type of testing; it did not validate the software against its original requirements, specifications, and design. Therefore, the final product could still be fault-free software (in terms of program bugs) that did not operate under the specifications in which it was originally designed. The bugs, errors, are classified here as three types, design, logic, and syntax (3:419). Studies show that of all software errors, design errors represent the largest percentage, ranging from 46 percent to 64 percent (3:419, 7:1230), and that these errors are usually not caught until late in the lifecycle, during acceptance testing or after

delivery (3:419, 55:150). These errors are more significant than their counterparts because they are the most frequent and the most expensive type of error. They are the most expensive because any effort based on the faulty design premises almost certainly will contain faulty material. The lesson is that errors breed errors, and those introduced early offer a greater potential for mistakes and require more work to be corrected. The ease of introducing errors multiplied by the magnitude of the modern software has created a general understanding that testing is now necessary at all phases of the lifecycle (19:170, 25:4).

Lifecycle testing begins with a test plan at the requirements definition phase and ends only when the software is no longer used or modified. At each phase of testing, managers are encouraged to produce test documents; this effort promotes more accurate testing (55) and generates a record of test results that can be review for consistency and completeness. Consistency is defined here to mean an internal conformity to software engineering standards and good programming practice; an example is that identical terms may not be defined differently in separate documents. Completeness is a property that the entire project holds at any particular phase; information is not lost when crossing phase boundaries or by rewriting the old information in a new form. When given this degree of

importance, testing becomes the the quality control activity which it was meant to be; however, it also generates test documentation which requires additional effort.

To reduce the "extra" burdens of testing, managers and software engineers have sought to automate much of the testing process and its documentation. This effort has resulted in the creation of requirements definition tools, test case generators, "smart" compilers, debuggers, and many other static and dynamic testing tools. These tools support the necessary lifecycle testing activities. Yet to be determined is what subset of this "grab bag" of diagnostic tools would be most effective in a software development environment.

In 1980 the National Bureau of Standards (NBS) began to prepare a guideline to help improve software quality and identify weak areas in verification and validation, V&V, testing (56:373). Their assumptions were that a number of good V&V tools already existed, but there was not yet a clear definition on how they might be integrated into a single useful lifecycle test tool. Their solution was to create a V&V guideline that would present essential V&V methods which could be used for any specific project (56:367). This investigation operates under the same premise that the tools already exist and that they need to be integrated, but the focus is restricted from the



generality of a guideline, to a specific design for a real-world environment. This environment is the Air Force Institute of Technology's Software Development Workbench (AFIT SDW). The design pinpoints the types of tools which are useful to each phase which the SDW implements or is projected to implement. It also shows how the selected diagnostic tools are integrated to support the SDW. This initial design is then refined and detailed to complete the design portion of this investigation. The purpose of this investigation is to continue support for the SDW as a state-of-the-art programming environment.

#### Scope

The intent of this investigation is to carefully and completely analyze the testing, verification, and validation needs for the Software Development Workbench (SDW). A system design reflects this analysis, and although it is derived from specific SDW's needs, it is, in general, applicable to similar programming environments which use the six phase lifecycle model. When this model is not used the distinction at the various levels may be lost, and yet the majority of the sequence and the tools which are suggested should remain valid. The system design is refined further to reveal diagnostic tools and procedures which support automated testing in an interactive environment. The implementation of this design provides the control structure

for the support of these test tools and procedures. Additionally, as many of these tools and procedures are added to the control structure as time permits.

#### Assumptions

The nature of this investigation requires coordination with AFIT's SDW. The SDW is currently implemented on a VAX 11/780 under the VMS operating system.

#### Summary of Current Knowledge

Considerable effort has been devoted over many years in developing methodologies and tools for testing; however, the major thrust of this effort has concentrated on the specific phases of the lifecycle rather than the lifecycle as a whole. Thus consideration for lifecycle methodologies and an integration of tools for entire software testing has been overshadowed by the enthusiasm to produce new diagnostic tools. The applicability of those tools and their interfaces with other tools still needs to be defined. This does not suggest that no work is being done in this area; this is not an empty field (20, 31, 51, 56), and the effort should continue.

#### Standards

Standards are a necessary part of any research and development effort. They are the metrics that impose a specified level of quality in the report and during the investigation. They also assist the investigator by

proposing the outline in which his material is presented. Finally they aid those reviewing the work by specifying the "frame of reference" for particular sections of the report.

The standards for this investigation are from the following resources:

- 1) Style Guide for Theses and Dissertations (17),
- 2) Thesis Projects in Science, Engineering (13),
- 3) IEEE Standard Glossary of Software Engineering Terminology (4),
- 4) Structured Analysis (60), and
- 5) Elements of Programming Style, Second Edition (36).

These resources provide the standards for the report format and the material presented herein. The thesis investigation is guided by presently acceptable and AFIT-taught software engineering practices (48, 60, 68).

#### Approach

This thesis begins with a literature search of the topic. This search continues throughout the majority of the thesis investigation so that a complete investigation of this topic's information is achieved. The search covers three major areas:

- 1) materials which illustrate the demand for lifecycle testing,

2) materials which describe requirements and standards for the various lifecycle phases, and

3) Materials which group tools in a particular phase and suggest methods of interfacing those groups.

The requirements necessary to realize the thesis objective are presented in the following chapter.

These requirements are drawn from the analysis of the material examined during the literature search and in consideration of the SDW's current configuration. Included within the requirements phase is a functional model of how the requirements are satisfied. This model is proposed as a launching point for the preliminary design of the SDW test methodology.

The preliminary design defines how the requirements are met by suggesting the types of tools that might be useful at different phases of the lifecycle. Also provided in the preliminary design are brief discussions on each of the tool types highlighting their usefulness. The types are further defined in the detailed design phase by a careful analysis of the specific needs of each phase and how the tools can best satisfy the functional model in the requirements definition. A best method of implementation of these tool sets is chosen. "Best" is determined by the degree in which the requirements are satisfied, support of the requirements currently defined for the SDW (25), and agreement with the

standards of this investigation and AFIT.

The detailed design considers not only the needs of software testing in general but also the demands that now exist for enhancements on the SDW. Consideration is given to maintain the modularity of the SDW; the goal is to produce an enhanced version of the SDW without destroying its structure or creating a "tangled mess" of code.

The coding is directed by the design and by current practices in structured programming. At this point, full familiarity with the procedures for producing code on the VAX is attained. The implementation language is chosen as a reflection of the author's capabilities, and of the extent to which it will aid in the interfacing with the SDW and any additional tools.

Integration of the new code and the tools with the SDW along with the acceptance testing finish the operational portion of this investigation. During this time, the acceptable test cases are documented for use in regression testing. Also the report and the software's operating appendices are completed.

## Materials and Equipment

The SDW was developed and is currently running on AFIT's Digital Engineering Laboratory VAX. For continuity of development and because of its availability this investigation requires the use of the DEL VAX and its peripherals.

## Summary

It is the intent of this thesis to examine, in depth, software testing needs over the entire software lifecycle. Extensive research in the testing process and its relationship to the lifecycle is done to insure complete coverage of the problem as it is presented. The outline of project also follows the lifecycle approach.

## Requirements Definition

### Introduction

The requirements definition phase is the beginning of the software lifecycle. In this phase, the needs that must be satisfied are outlined from a users point of view. The needs are stated in the form of requirements and are expressed functionally in a requirements model (54). This functional model states what is required and why (7:1227). The Requirements Model (the what) is also the launching point for the Preliminary Design (the how) which is the next phase in the software lifecycle.

There are a number of ways in which the needs of a new system are established (54); however, there are three elements which are common to the development of software systems: originality, applicability, and developer/user interface. Originality refers to the functions which will be automated by the proposed software. If the functions which the software will perform are revolutionary, this means they are not yet implemented by any method manual or otherwise, then the designer must "brainstorm" on what the requirements should be based on his experience with systems which perform similar functions. In many cases, the proposed software is for replacing or enhancing an existing system. In this instance originality is low, but the

also remember that the requirements definition phase is not bounded by implementation concerns. This does not imply that reality is not important, but that the requirements should honestly state the user's needs with very little regard to how they will be implemented. This detachment from the environmental constraints is the heart of the brainstorming activity, which allows the developer and user the freedom to express their ideas. The goal is that these ideas will truly reflect the objectives of the software and that they will remain bounded at this point only by reality. Since this investigation is a continuation of the Hadfield thesis (25), it most closely resembles the model of enhancing an existing system.

#### Testing Needs

All engineering disciplines require some type of quality assurance program. Software engineering is no exception. The complexity of software demands that it be tested thoroughly for accuracy and reliability. There are a number of ways to do this testing (41). Popular techniques used by the DOD and other civilian corporations are in-house quality assurance (QA) programs and independent verification and validation (IV&V) (41, 50). Experience has shown that software QA and V&V are necessary, because the nature of software production allows people the opportunity to inject errors during the software's development (43:78).



requirements must still be identified. If the documentation addressing the existing system's requirements is available the designer should read it carefully for understanding:

- 1) the objectives that the original system was supposed to meet,

- 2) to what level did the original system meet those objectives,

- 3) how the objectives might have changed or have been modified since their initial identification,

- 4) what objectives were not satisfied by the original system, or what objectives ceased to be met because of environmental changes, and

- 5) what new objectives, if any, have arisen and require their own functional components.

Applicability refers to the "real" need for a software implementation. Often the functions to be implemented are already being performed in some manual way. The software engineer and the user must together decide whether a software system is the answer and if it is, then what its scope should be.

When the software is designed for a particular user, the software engineer must interface with the user concerning the particulars of the software system. If this developer/user communication is ignored then the software engineer risks defining inappropriate requirements.

As the software engineer determines the requirements for the software with these three things in mind he must

The differences between testing, QA, and IV&V are not always clear (figure 3). If the software project is sufficiently small there may be no differences at all in these activities. When this is not the case the differences are apparent. Testing is the activity performed by the software engineer which should insure the correct operation of the software from a module to the system level. QA is an activity performed by an in-house group. It is essentially the same as testing, but it is done independent of the developing engineers. This independence from production allows the group to strictly enforce quality assurance standards and to test the software against its requirements

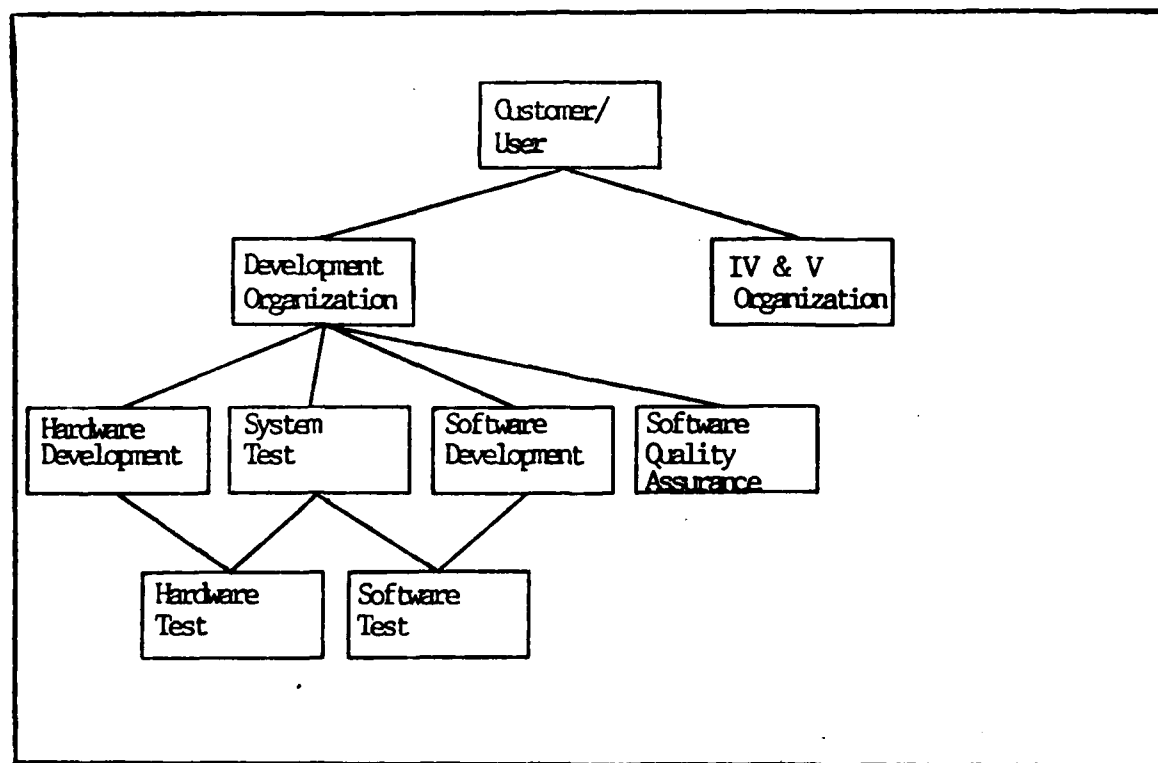


Figure 3: Organizational Relationships of QA and IV&V Teams  
Source: Lecture Slide, AFIT EE545 Course

more objectively than the developing software engineer. QA also evaluates other software activities such as configuration management and documentation support (53:66).

IV&V is an activity performed by a organization independent of the developing organization and at the request of the customer or user of the production software. IV&V benefits from a "distant" view of the software, another angle, independent of the company policy of the production organization. These three activities when they are well managed by the purchasing authority create a "quality net" which will catch some of the errors before they become critical or expensive. This whole idea of assuring quality by different testing organizations is simply a good idea with today's software performing more critical functions (e.g. tasks upon which human life depends). It is similar to "getting a second opinion;" the confidence increases if the opinions match.

It was mentioned above that these three activities, testing, QA, and IV&V, are not always distinct. This is typical at AFIT where the software is almost always produced by groups of four or less. In the AFIT environment the team of software engineers is the entire development team; therefore, a QA team is not possible, yet QA is still necessary. The IV&V organization is never requested because there is no management structure to hire them, yet

verification and validation V&V are still important. Some how these activities must be included in the development process, even in small projects, to insure reliability and to give confidence to the software. This is one of the goals of software development environments (15, 25), and it is a central concern of this investigation.

#### Error Types and Tests

Software engineering is a human production process; this makes it an error prone process. The purpose of testing is to remove the errors which will inevitably occur. Before the requirements for trapping these errors can be identified, the types and the costs of each type of error must be understood.

There are three distinctive types of errors, they are design, logic, and syntax. Design errors are introduced during the pre-implementation phases of software development or the preliminary and detailed design phases (figure 4). Examples of design errors are an improperly leveled program structure or a lack of modularization of the code. Logic errors occur during the detailed design and implementation phases; they are classified as logic errors, because they result from improper logical thought. Roundoff errors and looping errors are typical examples of logic errors. Syntax errors are violations of a particular programming language's grammatical rules. Syntax errors can

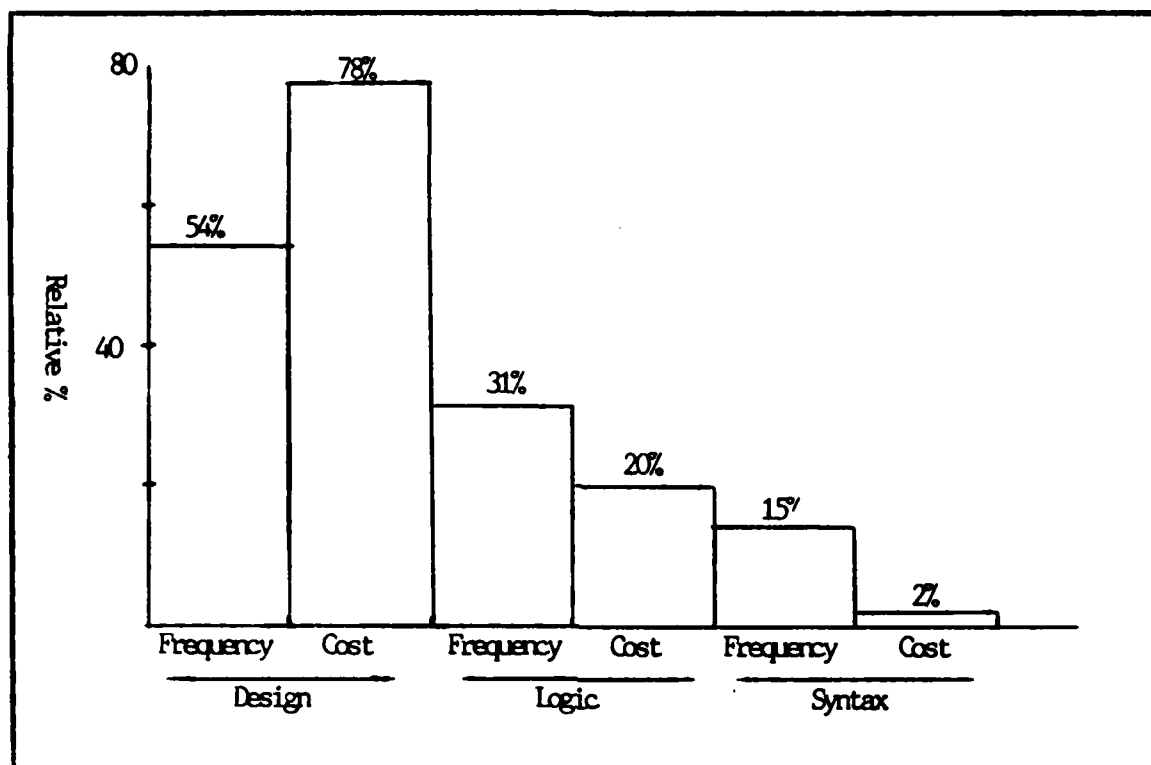


Figure 4: Error Frequency and Cost

occur whenever the problem solution is written down according to formal language rules. A language symbol or term improperly placed or missing in a solution statement is an example of a syntax error (e.g. a misplaced or missing semicolon in a pascal statement).

Each of the error types discussed above has their own relative cost for correction. This relative cost is determined by comparing the average dollar amounts to correct an error of each type. A primary factor which affects the error cost is the elapsed time between the error's introduction and its discovery (figure 4). Because syntax errors are discover relatively quickly after the are

introduced they are the least expensive of the error types. In contrast to the syntax error, the design error is the most expensive type of error to correct, because the time lapse between introduction and discovery can span up to four lifecycle phases. The significance of the elapsed time is that much of the time spent in the detailed design and coding phase might be wasted because the work done in those phases was based on a faulty premise in the preliminary design. Also, design errors tend to be more encompassing; they usually affect a larger segment of the software than either the logic or the syntax error. The logic errors have a cost that falls between that of the design and syntax types depending on the portion of software they affect and their subtlety.

Based on the cost alone a decision could be made to concentrate on finding ways to stop design and logic error early. Additional motivation for this decision are studies which show that of the error types, design errors represent the largest percentage (4:419). With this information in mind a few objectives concerning testing can be identified.

- 1) testing must begin with the beginning of the lifecycle to insure proper testing for all error types at all phases,

- 2) an effort must be made to reduce the time from when an error is introduced and to when it is discovered, and

3) emphasis should be placed on insuring (as best as possible) that the design is "good".

Minimizing the time to discover an error is the key to effective testing. The methodology developed in this investigation attempts to satisfy that need. At each phase, what must be tested changes; therefore, a lifecycle approach to testing must be employed. The methodology presented herein acknowledges these needs and objectives and provides the capability as well as the management for lifecycle testing.

#### Requirements Considerations

In determining how reliability and confidence can be better integrated with the production of software a few elements of the proposed environment must be considered. The first is defining the size of the group that will produce the software. This investigation is concerned with the small project group, less than five. In this situation all the group members have expert knowledge about the software and its requirements. The size of the group discourages parallel design efforts on these requirements in search of a "best" model; therefore, after some discussions on the needs and concerns of the system the group accepts a single model to meet the requirements. Because of each member's familiarity with the project unbiased opinions about the software quality are not a realistic expectation.

Software quality is heavily dependent on how accurate the requirements are defined (54:139). It is necessary then that the environment maximize the capability to produce clear unambiguous requirements which will assist the group in making the correct decisions for the model, even with a minimum of guidance from sources outside the group.

The second element is time. The frame of reference in this investigation is a maximum of nine months from the conception of an idea to delivery, and this time is often much shorter. With the small group size and the short time span, testing must be carefully managed, or the group may simply run out of time before the project is completed. The lack of time also inhibits the development team from taking time to learn each development phase's testing needs, and because software development in the proposed environment is largely independent, there is a limited amount of expert knowledge in testing from extra-group sources. Additionally, the small group prohibits them from dedicating a team member as a full time testing manager. Thus, the software development environment should integrate testing as a part of development; adding instructions and guidelines to assist in the test management. This methodology leans towards an algorithmic approach of developing software to reduce indecision on how to manage small projects (without management), or on how to progress to the the next phase of



development from each preceeding phase.

A final consideration is resources. Generally software in the proposed environment is developed with the tools on hand. This requires a software development environment which will support the development of a variety of software projects without add-ons and extras. Stated another way, this investigation is interested in a complete system; complete is defined to mean that each phase of development and testing can be executed on the development system as it exists. These considerations help to scope the direction of this investigation, they reflect the AFIT SDW environment, and they also help to answer how reliability and confidence can be integrated into the production of software, in the SDW environment, by the unification of development and testing in a single environment.

In this investigation "SDW environment" refers to all environments similar in characteristics to the SDW environment; these characteristics are a small (four or less) software development group and a short (nine months or less) development schedule. The SDW environment was chosen because of its availability, and because of the authors experience with a few of the SDW components and host operating system. Another consideration was the desire to enhance the capabilities of the AFIT system by the students and faculty at AFIT. A final consideration was the

educational opportunity to enhance a large system while satisfying all former requirements, and meeting new requirements to produce a integrated development and testing environment.

#### General Requirements

Unifying testing and development in an interactive automated environment is the key to a complete testing methodology in the SDW environment. Unification simplifies the management of the project so that the small group can understand and control each phase of development and testing. Automation relieves the group members from many of the tedious and repetitive tasks often associated with testing and its documentation, and allows them to focus more on critical areas of development and testing. However, there are activities which an automated environment cannot support. IV&V cannot be automated because by its nature it is independent of development. The alternative is to automate V&V , which is still a necessary function, as part of the QA requirements. This is acceptable if the software developers take note that there is now no "outside-the-system" checks on the development. A functional model at the end of this chapter explains more explicitly and in greater detail the needs of a SDW type of environment with respect to software testing. These needs are presented below as general requirements.

Software development and testing go hand-in-hand and work best when integrated together throughout the lifecycle (2). Hadfield's thesis, (25), defined the initial structure of the SDW; therefore, a preliminary objective of this investigation is to meet the requirements imposed by (25). A second objective is to make this investigation applicable to a variety of environments similar to the SDW environment and using the software lifecycle for program development. This necessarily requires that the requirements definition phase of this investigation be somewhat independent of the SDW implementation. From this vantage point a new set of requirements are established to integrate with and support those of the Hadfield thesis and to address the independent concerns of software testing during the lifecycle.

Hadfield recognized the importance of testing when he documented the requirements for the SDW (25:39), and in support of those needs he included diagnostic tools as a part of the SDW implementation. But this legacy of tools has no guidelines and no motivation for their use in the development process, except for their presence. This chapter presents the requirements necessary for the integration of these diagnostic tools, and other testing tools which are defined in the design chapters of this investigation.

The requirements stated here are properties that are deemed necessary or desireable in the testing process. These requirements were identified by experts in this field (18, 34:144, 45, 55) and by the author through an analysis of the introduction of errors in actual programming projects spanning the entire lifecycle. As a complete set, these requirements idealize the testing process. The intent of this investigation is to create a model that satisfies these requirements, and to design software that functionally executes these requirements under the implementation constraints of the target environment. The following is the established list of requirements which the test methodology of this investigation should satisfy :

- motivational
- developmental
- timely
- understandable
- organizational
- managerial
- automated
- self-documenting
- traceable
- flexible
- language-independant

The order of the listing does not imply the degree of importance of each requirement. In different environments some requirements will be selected as critical while in others they may only be desirable. In this investigation the relative importance of each requirement is discussed during the preliminary design phase when implementation issues begin to arise and tradeoffs between requirements can occur.

Motivational. In order for a methodology to be useful it must be used. As previously discussed, the software community realizes the importance of testing and validation, and yet unless the realization results in testing activities by the software engineers, then the theme of testing is lost. Testing should be motivated from the beginning by requiring positive evidence of quality and reliability in each of the lifecycle phases (55:150). Therefore, the tests used in the test methodology should be able to provide the software engineer with the ability to test in all phases and provide the means for presenting the evidence of quality and reliability.

Developmental. Often testing is not well received because it is seen as "extra" work. The prevalent attitude is to get the program running first and to test it "if there is time" (55). Too often testing is viewed as tearing down the software which is the product of considerable time and

effort. Colloquialisms such as, "running the program through the mill", are common during testing, and they reflect the negative attitude towards complete lifecycle testing. If testing was developmental, a process that assisted in production, and it was not seen as destructive or an "extra", then testing would be a desired active process. By selecting testing tools for the SDW which are profitable for development while at the same time appropriate as diagnostic tools respective to a particular lifecycle phase, the development aspect of testing is increased.

Timely. As mentioned earlier, software tests must be timely to be accurate and the errors inexpensive. The expense of an error is largely dependant on how long the error remains in the system; therefore, if diagnostic tests are applied at timely intervals during the lifecycle, then the event that an error will remain for an extended amount of time (more than one phase) is reduced. For this reason the methodology is required to support testing at all phases of the lifecycle.

Understandable. It's not easy to hammer a nail with a wrench, but if the purpose of the hammer is obtuse, then the wrench might be the "best" tool. Often software tools are not used because they are not easily understood (38:5). In the SDW environment, time is not available to learn

complicated diagnostic tools, even if they provide much needed functions. This time crunch influences the development group to consider only short term benefits; therefore, they need to be able to rapidly evaluate the capabilities of each tool to judge whether it will be useful in their application. The key to rapid evaluation and judgement is understandability. Understandability is a necessary requirement in the SDW environment, and it is desirable in any environment where time is an expense worth reducing. Rapid evaluation of each tool's capabilities gives the software engineer the knowledge to use each tool in the application for which it was intended.

~~Managerial~~. The managerial requirement is necessary to give direction and guidance to the testing process. The managerial approach provides an outline for the phases of testing much like the lifecycle outlines the phases of software development. By partitioning the test methodology into phase sets, the management of testing is organized and simplified (18:122). The "road map" management of the methodology provides the software engineer with guidance on how to proceed in testing, but a drawback to this approach is the opportunity for the software engineer to think the job is finished once the final phase is completed. This is not always true since the test methodology is an outline for testing and not a formula for testing. Software can

provide a pseudo-manager in its control structure (this is a goal of the methodology), but the software engineer must decide whether the testing is actually complete.

Organizational. Hand-in-hand with the requirement for management in organization. Indeed, unless a project is organized, it cannot be efficiently managed, neither can it be automated. Organization can be applied to all areas of software development, and should be, but the focus of this investigation is to organize testing by requiring that an organization of these three elements of testing:

- 1) the classification of test functions,
- 2) the format of the development data, and
- 3) the format of the output data.

By organizing the tests into functional classes of application (e.g. tests for consistency), the software engineer can be reasonably certain what facets of the software have been tested. Organizing the development data is necessary if the tests on that data are to be automated; likewise, the output must have defined formats. However, the intent of this investigation is to proceed beyond simply defining a format for the diagnostic output, but to define a format that in its presentation is also an aid to the software engineer in evaluating the the software's sucess in satisfying the methodology's tests.



Automated. Automation is the key to lifecycle testing. In the SDW environment automation relieves the development group from documentation and tedious QA tasks and permits them to concentrate on other areas of development and testing. Automation of testing throughout the lifecycle enables development and testing to be integrated into a single software development system, thus simplifying management (18:122). More importantly, integration into one system allows testing and development to use the same database.

The significance of using the same database for development and testing is far-reaching. Errors introduced when interfacing between test and development are eliminated, tracability of information through the lifecycle is enhanced, and the flow of development is channeled simplifying organization. These claims are possible if one realizes that the representations of the software under development, at each phase, are modifications of a previous development phase reformatted (detailed) to satisfy representational criteria. Once these transformations are formalized, the appropriate tests of the information for each phase become somewhat algorithmic (59:78; 61:902).

The automated test methodology described in this investigation includes on-line guidance to help the user select appropriate tests. It interfaces the development and

the diagnostic software through the database to provide consistency between development and testing. It also generates the associated test documentation for development history and for error analysis by the software engineers.

Self Documenting. Documentation is the record of software development. Without this record, the software would be useless, because software exists only in its documentation (53:24). Documenting activities and results as they occur provides the most factual and useful information about the development possible (31:156). If the documentation is delayed, the opportunity to forget pertinent details increases. During analysis and testing reports on the software responses are necessary for accurate decision making and for the traceability of errors. If these responses can be recorded automatically then time and test documentation integrity can be saved (28:101). Over the course of development modifications to the software will occur and they must be documented because they will affect such things as interfaces, functions, or timing; these things being important to individual test cases. For the same reasons test cases are documented for use in the maintenance phase as regression tests. Interactive automation of this documentaion allows for more timely, accurate and complete information.

Traceable. In requirements definition phase the conditions which the software must satisfy are presented. Traceability means that the requirements presented in the requirements definition can be followed through each phase so that at a functional level they can be tested for satisfiability. Traceability may also be reversed; low level functions may be traced backwards to the requirements which motivated them. This capability to trace backwards is important, because it enables the software engineer to analyze the original conditions and constraints which create problems. Traceability adds understanding and confidence to each new phase by justifying the new development with identifiable relationships to previous well tested phases (31:74).

Flexible. The SDW was designed to support numerous development applications (25:36). If this original objective is enforced, then the test methodology to be integrated with the SDW must be flexible in order to maintain the SDW as a state-of-the-art development environment. All applications supported by the SDW are supported by this methodology.

Language Independence. Language in this investigation refers to the high order language (HOL), e.g. Pascal, FORTRAN, C, which the software engineer will choose for the design implementation. Language independence allows the

software engineer to have freedom from selecting an HOL until the design phase is nearly complete. A late choice of an HOL enables one to match the characteristics of the software with the language by providing the maximum amount of time to analyze detailed design information and appropriate HOLs. The compatibility between implementation and language promotes ease and efficiency in the coding phase.

#### Requirements Representations.

The final step in the requirements definition phase is the construction of a functional model to express the preceeding requirements explicitly. There are a number of ways to represent this model. They range from structured english representations, such as Program Design Language (PDL), to graphical representations, such as Data Flow Diagrams (DFDs) or Softech's Structured Analysis and Design Technique (SADT). Each of the methods has advantages and disadvantages in the manner in which they represent requirements.

PDL is useful if the general system structure is already understood and it can be statically executed for syntax, but it does not show data or control flow, nor does it present processes in an explicit manner (8). In addition, PDL's similarity to HOLs may tempt the software engineer to overlook the design phase in favor of coding.

DFDs and SADTs are two similar graphical techniques for representation of requirements. Both explicitly show processes, data flow, and heirarchy; although, DFDs do stress data flow while SADTs stress processes or activities. DFDs have less notation than SADTs and are generally simpler to understand; nevertheless, SADTs were chosen to represent the requirements in this investigation for the following reasons.

SADTs were chosen primarily because they are supported by the SDW as an automated tool, and because they are a standard technique for representing requirements at AFIT. Other considerations were the undestandability associated with graphical techniques, and the author's prior experience modeling requirements with the SADT diagrams. It should be noted that there are many other techniques for representing requirements that were not presented here, because they were not considered (6, 27, 29, 57, 62).

The SADT basic notation is a rectangular box denoting an activity and arrows showing data and control flow (52) (figure 5). The arrow's respective positions around the box determine wether the data labled on the arrow is control, input, or output. Arrows may also represent mechanisms, which support the SADT activities to which they point. Activity boxes are connected together by data and control flow; sequence is implied by the order in which the boxes

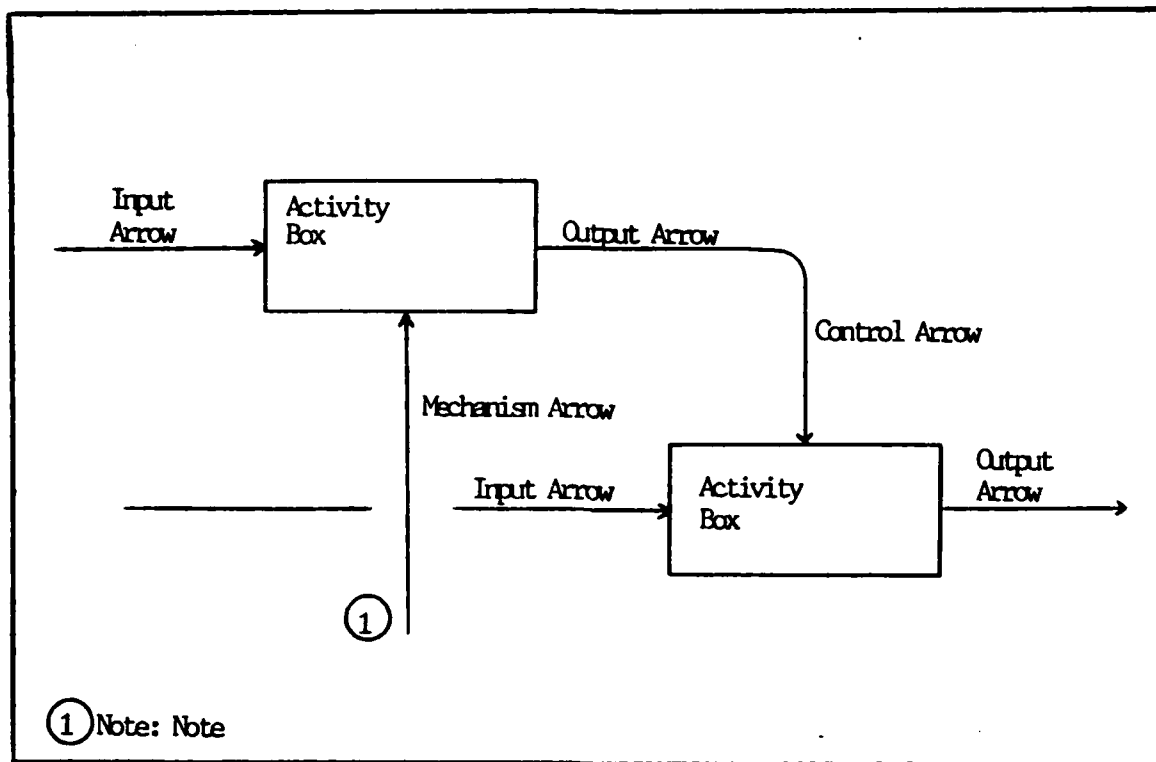


Figure 5: SADT Symbols

appear on the SADT diagram, beginning at the top left corner and moving diagonally to the bottom right. Readers not familiar with SADTs are encouraged to review material by Peters (48) or Ross and Schoman (52) for more detail.

#### Functional Model

The functional model outlines the requirements for the SDW Test Methodology. These requirements represent the testing needs for the entire software development lifecycle, and they are influenced by the desire to integrate this methodology with the SDW to produce an integrated test and development environment. The model represented by the SADTs includes the textual information necessary to understand the

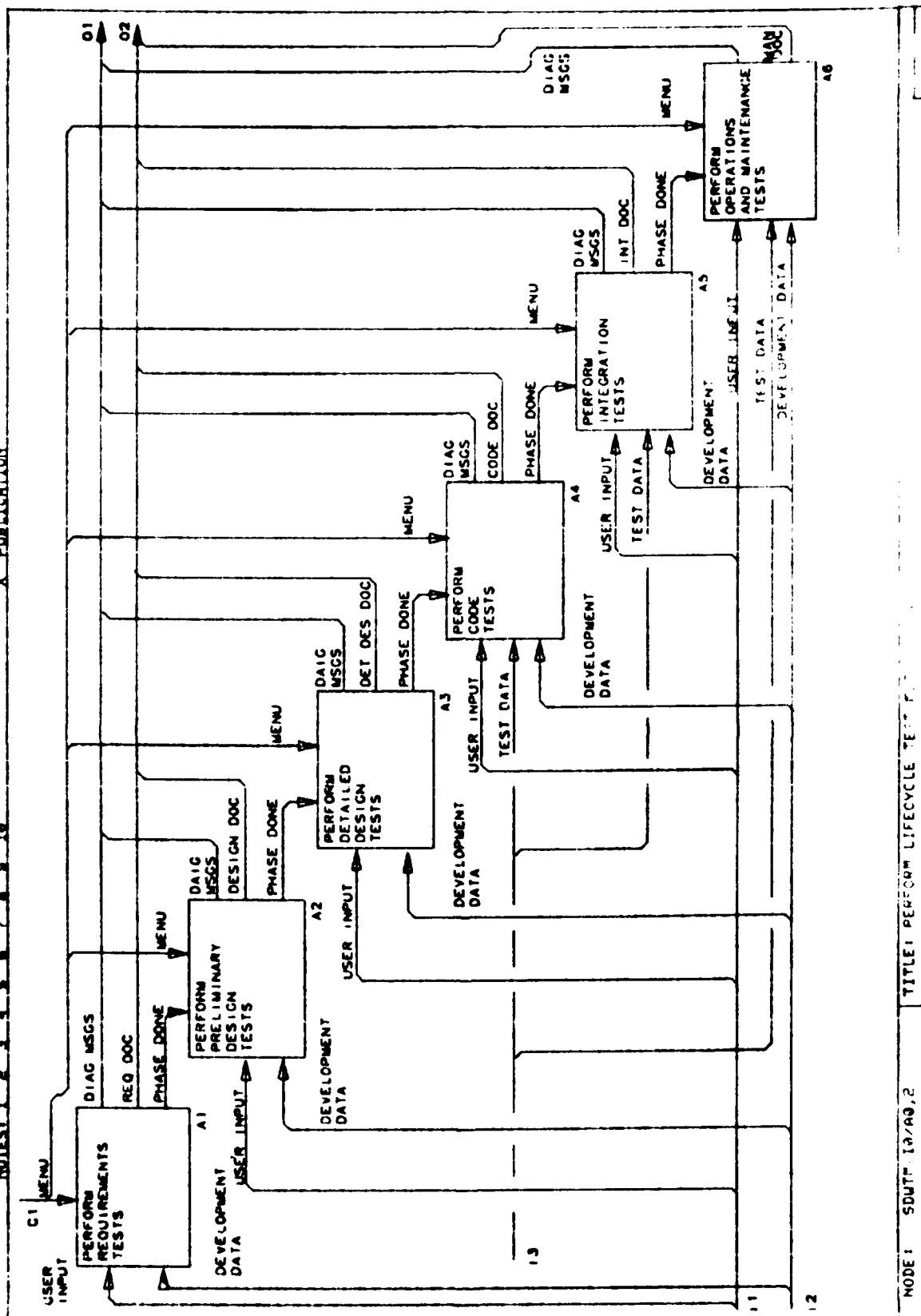


Figure 6: Functional Model A0 Diagram

particulars of each individual diagram.

The highest level, the A0 diagram (figure 6), of the Model is very similar to the software lifecycle presented at the beginning of this chapter (figure 2). The Test Methodology is divided this way to emphasize that each phase in the lifecycle needs to include testing. Additionally, each testing phase must provide the capability to document what occurred in that phase. This timely documentation helps to assure accuracy.

As each phase is further detailed, the types or categories of tests are revealed. In the requirements phase the types are correctness, clearness, and consistency. Errors belonging to these types are classified in this investigation by the same name; therefore, it is appropriate for an error to afterwards be referred to as a consistency, clearness, or a correctness error. Notice that this classification of error types is a subset of the larger classification of error types, design, logical, and syntax. The tests identified in the Model beyond this level of detail are presented as methods to trap their respective error types. Although, these tests represent only one of many possible solutions, the intent is that they are the best representation of a lifecycle testing solution in the AFIT environment. The complete Functional Model is presented in Appendix A of this investigation.



## Summary

The requirements definition is the beginning of the software development lifecycle. This chapter in the requirements definition for the SDW Test Methodology; it, in conjunction with the Functional Model in Appendix A, outlines the needs and objectives for the Test Methodology. It also lays the foundation upon which the Methodology's software structure is built.

Central to the development of the Test Methodology is the promotion of timely testing to reduce error cost. This idea results in the effort to insure lifecycle testing capability. Stress is also placed on the importance of automating test management and documentation to reduce the manual work associated with lifecycle testing.

Eleven general requirements are presented in this chapter and supported by the Requirements Model. The continuance of support for these requirements and the Requirements Model in the design and implementation of the Test Methodology is necessary to insure an effective lifecycle testing methodology.

## Preliminary Design

### Introduction

In the preliminary design phase the functional model of the requirements definition phase is mapped into a high-level design. This is a logical design which outlines how the requirements are satisfied; the logical design is detailed later during the detailed design phase in the next chapter.

The purpose of the preliminary design phase is to ease the difficulty of transitioning from the abstract requirement specifications to the detail of the detailed design by creating the high-level design as an intermediate step. This intermediate step allows the software engineer to concentrate on the structure of the proposed software and the function definitions without concern for implementation syntax. Careful attention to structure and definitions at this point eases the transition to detail, because it organizes and clarifies the software's purpose before the detailed design begins. This gradual refinement is known as top-down design (48:139).

Most methodologies for software design include top-down design in their application. Characteristics of top-down design are structured representation, decomposition, attention to module hierarchy, order of module calling,

concise names, and data flow representation. How these characteristics manifest themselves in the different methodologies is discussed later in this chapter, when the design methodology used for this investigation is presented.

The record of the preliminary design phase is the Preliminary Design Document; this document corresponds to Chapter Three of this investigation. The Preliminary Design Document presents justification for the preliminary design in a variety of informational formats. Information particular to the target environment, a configuration model, a discussion on the resolution of requirements, and the preliminary design comprise the sections of the Preliminary Design Document which support and justify the design. This chapter contains the first three sections; the Preliminary Design and its supporting text are presented in Appendix B.

#### Preliminary Design Concerns

During the preliminary design phase the software engineer begins to be concerned with any constraints the environment might pose. However, despite these constraints, the software engineer should avoid designing a system which would be useful in only the proposed environment (e.g. extendability). This implies that the designer must consider information other than the requirements definition documentation when producing the preliminary design. The following is a brief list of concerns that the software

engineer should consider prior to the construction of a design:

- generality and alternatives for similar situations,
- budget (time),
- technology (resources),
- development team's capabilities/potentials, and
- environmental and user constraints.

Environment and resources are issues with related meanings, and they can appear to be the same thing, but as used here, the environment is the state of the development area which generally may not be changed, while the resources are those materials which support the environment.

Software is dynamic in that changes occur throughout its life due to corrections, modifications, and enhancements. Software is also expensive as demonstrated in figure five, in the previous chapter. Therefore, it should be designed for long life, and this requires that the software be flexible so that it can handle the modifications which will occur over its lifetime. Software designed only for a particular user and a particular environment will quickly become outdated. If the software engineer carefully considers similar applications, generalizes them, considers alternatives, and designs from this informed perspective, then the design will have the proper background to support the proposed software and future modifications. Considering

alternatives and future development is simply smart design.

Although the design should satisfy all the requirements presented in the requirements definition, this expectation may be unrealistic. Some of the concerns listed above may require trade-offs. One of these which has its affect on the design is the budget. The software engineer must design a system which can be implemented, integrated, and maintained within a cost outline provided by the user. Because time can also be considered a cost, the design must meet the temporal as well as the dollar costs.

Another factor is technology and resources. The design must be implementable under existing technology (sometimes "brand new" technology is used), and it must consider the resources available to the user to operate the software once it is delivered.

A final factor is the development team's capability and their potential over the course of development. The software engineer who produces the design ought not to design it "over-the-heads" of those that will implement it. Furthermore, the software engineer should look beyond development to the capabilities and the potentials of the software's user: perhaps a extensive help facility is necessary or a detailed menu system to help the user interface with the software.

Considering the user is necessary in software

development. This is not only good from a management point of view, but such consideration usually helps the software engineer understand the problem better. In addition, the user may require special features or functions which curtail alternatives. There may also be constraints imposed by the environment such as the target machine.

All five of the concerns previously listed are closely associated in that tradoffs in one area might produce "slack" or increased capability in another area. The software engineer's responsibility during the preliminary design is to weigh these tradeoffs, if they exist, and produce the "best" design according to his capabilities. The next subsection presents each of the five concerns with respect to the SDW environment.

0 • Preliminary Design Concerns in the SDW Environment. The Software Development Workbench (SDW) is designed to meet the needs of the AFIT software engineer. The AFIT software engineer is defined as students or faculty at AFIT who engage in the development or maintenance of software or participate in the software engineering courses. Although the SDW was designed with a particular user in mind, it provides the necessary elements for most general programming environments (25:40). In this manner it is applicable to a variety of development applications. The SDW Test Methodology is also design to aid the AFIT software

engineer, but while supporting a test methodology general enough to be applicable to other software development environments. As previously discussed there exist other concerns which affect the design.

Budget in this investigation does not refer money, but to the amount of time which is available for development. There are two reasons for this definition. 1) Many of the financial decisions are resolved before the project begins and students are encouraged to use the resources already provided. 2) Budgeting time is often cited as a problem over the course of development in AFIT software projects (13:30). Considered in this investigation is the notion that accurate and complete testing takes time. Therefore in order to motivated the use of the SDW Test Methodology, which competes against other software development activities for the software engineer's time, emphasis is placed on integrating the methodology with the lifecycle process and automating it as a component of the SDW.

The SDW Test Methodology is to be initially hosted on the AFIT Digital Engineering Laboratory VAX 11/780, the host machine for the SDW. As a consequence, the Test Methodology is design to be hosted on a "mainframe" size computer. This restriction is eased somewhat by the modularity of the SDW Test Methodology which allows portions of the methodology to be used on machines with less storage and computing power.

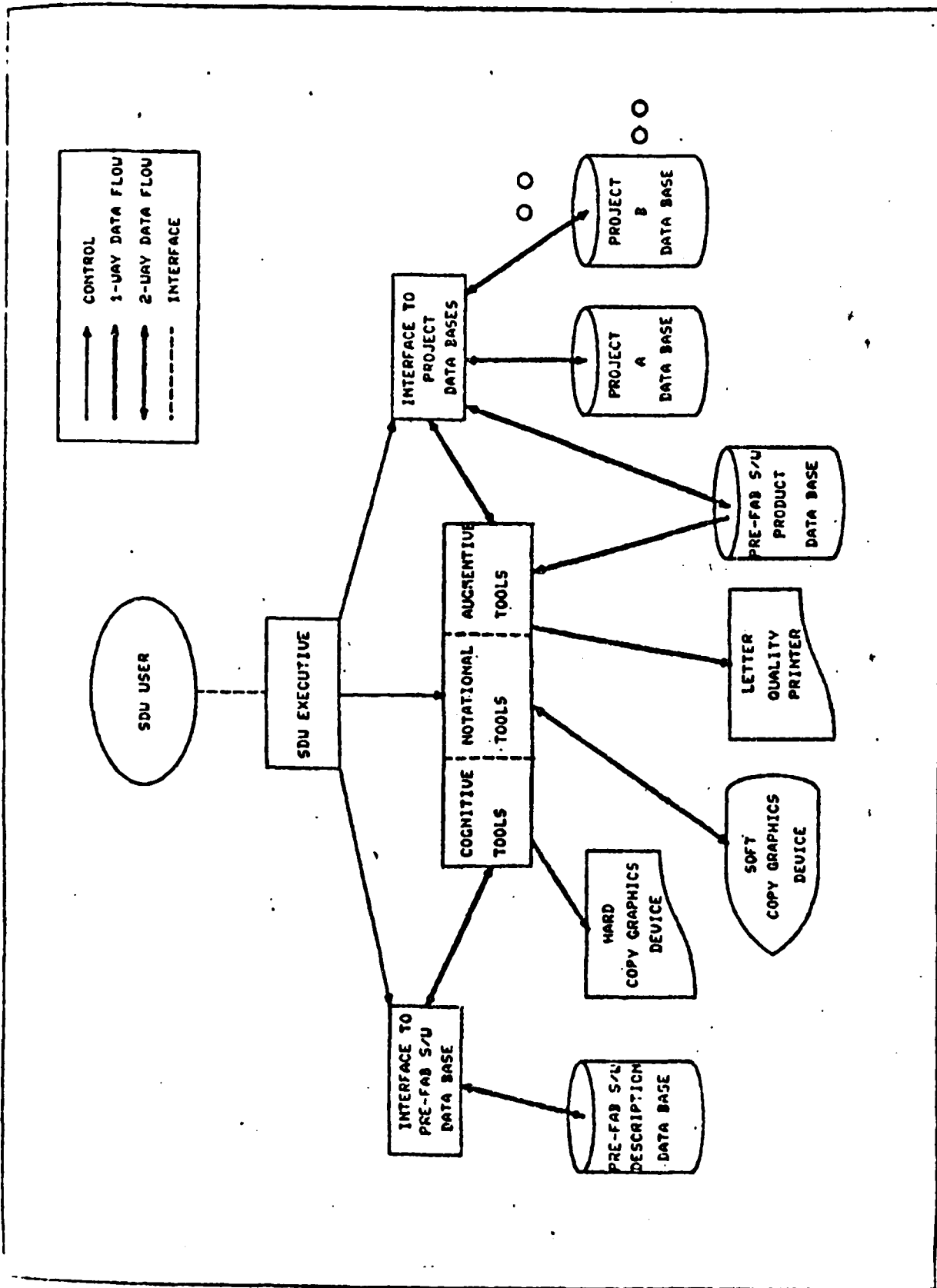
The development team is defined here as a student group of no more than two members or members of the faculty engaged in software development. These people reflect a well educated technical audience. The Test Methodology is designed to interface with this type of audience. Since the AFIT community is a rather nebulous "user" there are no specific user constraints except those determined by the author and the advisor of this investigation. The SDW Test Methodology is voluntarily constrained by attempting to satisfy the testing concerns outlined for the SDW (25:111).

#### SDW Test Methodology Configuration Model

The purpose of the Configuration Model is to identify the software system components which are necessary for operation of the software. The Model also explicitly outlines interface, control, and data relationships between the hardware, software, and other system components. The Configuration Model for the SDW Test Methodology, shown in figure 7, was originally proposed by Hadfield in his development of the SDW (25:96). The description and justification which accompanied the model are included in Appendix E of this investigation.

Defined within the model description are the purposes of each of the three types of tools supported by the SDW, the "cognitive tools", the "notational tools", and the "augmentive tools." Brief definitions are given here for





SDW Configuration Model

Figure 7:

each of the tool types to aid the reader in the succeeding discussion on the configuration of the Test Methodology with the SDW. The cognitive tools are educational and support current software engineering techniques. The notational tools assist in the production, modification, and documentation of development text and data. The augmentive tools provide the test and evaluation functions required in software development.

The Configuration Plan. Since the Test Methodology is itself a tool to support software development, the Methodology is integrated with the SDW under the management of the SDW Executive (SDWE). Although, the Test Methodology supports characteristic features of both the cognitive and notational tool sets (e.g. automated test management and documentational support), the Methodology is integrated under with the augmentive tool set, because software testing and evaluation is its primary function. Moreover, the Methodology further formalizes the loosely knit augmentive tool set by placing it under the Methodology's control and partitioning the tools in that set into functional classes. Integrating the Test Methodology under the direction of the SDWE allows the Methodology's diagnostic functions to access the development database; this provides simplified management of the database and an increase in testing consistency, because the test data need not be rewritten.

Directions for Integrating the Methodology with the SDW are provided for in the Software Development Workbench Executive Maintenance Guide (25:353).

#### Resolution of Test Methodology Requirements

Presented in the general requirements and the functional model in Chapter Two are the needs of the SDW Test Methodology. The answers to these needs are supplied by the Configuration Model, the Preliminary Design, and their associated text. This section explains how these needs are resolved for each of the general requirements by citing portions of the Design and the Configuration Model which support those needs. The Preliminary Design in Appendix B resolves the functional requirements model. Justification for the Design is presented in its associated text, its top-down construction, and the logical decomposition enforced by structure chart design.

Motivational. By integrating the Test Methodology into the SDW and under the direction of the SDWE, the software engineer is encouraged to begin testing early, just after the definition of requirements. Integration and the SDW database configuration permits the test tools to operate on the development database, and permits rapid accurate testing at each phase of development.

Developmental. The SDW test methodology is designed to operate in conjunction with the SDW and the lifecycle development model. Its integration with the SDW and adherence to the software lifecycle promotes early usage; this is important, because testing limited to the final stages of development, coding and integration, usually produces poorly designed and unsatisfactorily tested software (55:149). The tools included with the methodology help solve these problems and integration reduce the "extra burden" which testing is perceived to impose. Automated interactive consistency and correctness tests relieve the software engineer of much of the pencil and paper type checking. The automated documentation support provided further enables the software engineer to spend more time on the development and testing of software.

Timely. In order to keep the expense of errors low, the software must be tested at regular intervals during its development. This regular, or timely, testing helps insure that an error is not permitted to remain throughout several development phases. The Test Methodology supports timely testing by supporting testing in all phases of the lifecycle. Moreover, guidance is provided in the control structure to aid the software engineer in applying the "right" tests in any particular lifecycle phase.

Understandable. Understandability is necessary for the Test Methodology because it is the key to usefulness. By retaining the menu driven design of the SDW, the user need not learn new initialization commands or new system commands to execute the SDW Test Methodology. Development and testing on one system decreases complexity by allowing the user to rapidly become familiar and fluent on a single development system. Furthermore, an on-line help facility for testing is provided as an extension of the SDW help facility to promote understandability. These features combined with the modular design and the automated documentation support increase understandability by providing organization and management control to the software testing environment.

Managerial. As mentioned in Chapter Two, the AFIT environment is such that it provides very little project management support. For this reason, test management is integrated into the control structure of the Test Methodology. This management is first provided by the SDW help facility, and it is later provided by the Methodology's menu and help facilities. This type of "road map" management certainly cannot answer all the questions associated with software testing, but it does provide guidance in selecting and using the diagnostic tools which are available.

Organizational. Organization and management are complementary requirements, because organization is a by-product of efficiently managing a complex process. Promoting modularity by employing top-down design is one way to encourage organization. The SDW Test Methodology is organized with top-down design techniques and partitioned into the six phases of the software lifecycle. Classification is another way to organize. First, the tests are classified by their association to a particular development phase. Secondly, they are subclassified according to the functional classes identified in the Preliminary Design. Additionally, some of the subclasses are further segmented when the need arises. In this way those functions which are relevant to requirements consistency testing are grouped together as a requirements diagnostic tool; similar cases apply for each of the other development phases. Organization is also present in the Configuration Model through the classification of the tool sets (e.g. augmentive, notational, ect.), and the separation of the development information from the diagnostic tools into different databases.

Automated. Automation is provided by integration of the Test Methodology with the SDW. Integration permits the Methodology to be operated under the control of the SDWE, and it enables automated testing on a shared test and

development database. SDWE control makes the Methodology a keyboard operation, rather than a pencil and paper operation. A shared database eliminates much the retyping which is necessary in non-integrated environments to provide the diagnostic tools with test data. Automated documentation support is designed into the Methodology to assist the software engineer in timely accurate documentation of the software's status with respect to testing.

Self Documenting. Documentation supports software development, testing, management, maintenance, and history; therefore, it must be accurate, and it must be pertinent to be effective. Timely documentation salvages accuracy. To promote timeliness the SDW Test Methodology is designed to record the pertinent information soon after it is generated. "Soon" demands that the each diagnostic test take the responsibility for its own documentation support.

Pertinence is the second quality necessary for effective documentation. Associated with pertinence is the level of detail. Because intelligence and familiarity affect the need for detail, what may be pertinent as documentation for one application may be too much or too little for another. To compensate for the differences which affect the need for detail, software engineers can indicate the level of detail, before the documentation is recorded, according to their

needs. Because all of the documentation is stored in text files, as supported by the Configuration Model, these files may be edited to increase their effectiveness before they are produced as hard copies. The Configuration Model supports the printers and the hard copy graphics devices required for all of the documentation which the test methodology produces, and they are accessible through the the SDWE.

Traceable. Traceability is a necessary part of software testing, because it provides information which assists the software engineer in code debugging, system integration, and determination of requirements satisfiability. Traceability is provided for under the traceability and completeness tests which validate the development against previously tested phases. Validation is possible because the data for all the development phases is accessible to the test methodology through a shared test and development database. Traceability data stored in the database during a phase is used in the traceability and completeness tests of the next phase. Part of the traceability information is documented to assist in testing, particularly during integration and maintenance testing when low level modules are executed to test the satisfiability of requirements.



Flexible. Flexibility offers the means to test a variety of different software development efforts with the same methodology. The modular design of the SDW Test Methodology incorporates this type of flexibility by providing a range of diagnostic tools applicable to different phases and types of development. In addition, by integrating the diagnostic tools under the direction of the SDWE, additional tools may be added to support non-standard or atypical software testing. Because different tests may require variations in the mode of documentation (graphical vs. textual), the Configuration Model supports hard copy graphics and printer devices, as well as appropriate mass storage for the software generated files.

Language Independent. Because it is desirable to choose languages which are familiar, the SDW Test Methodology is designed to give the software engineer a choice of implementation languages. The SDW already supports language independence during the pre-implementation phases of development. The design of the Test Methodology reflects this support by providing diagnostics for these pre-implementation tools.

Currently the SDW supports only a few implementation languages, but other languages can be added if needed. Provisions are made within the SDW configuration for the addition of new tools; such tools might be translators

for new languages. Because the Test Methodology is integrated with the SDW, support tools for new languages can be added to the Methodology as easily as the new languages are added to the SDW. This capability is especially important as interest increases in the C programming language at AFIT, in the ADA language in the DOD, and in LISP and PROLOG in artificial intelligence.

#### The Preliminary Design Representation


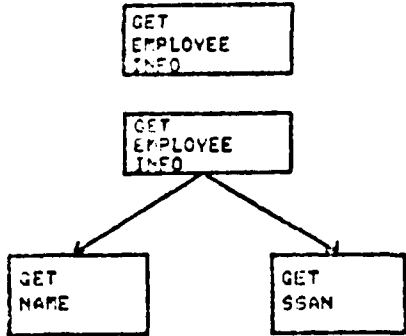



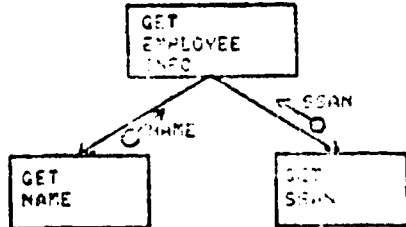
As mentioned earlier the preliminary design should add structure to the software. This structure can be presented by a number of different design techniques (8, 48, 49, 65) of these techniques, there are six elements which most hold in common. The six elements are graphical representation, hierarchy, order of calling, decomposition, input and output flow, and concise official names. The representation for this investigation, which was chosen from the structured design methodology (65), explicitly displays all six of these elements; this representation is structure charts. Structure charts were chosen because they included all six elements of a good design representation; moreover, they have the following qualities which made them especially desirable:

- 1) Among high-level design techniques structure charts are one of the most common, particularly at AFIT where the structure chart is a pseudo-standard in design.

2) As a graphical technique the structure chart displays hierarchical information in easily read diagrammatical form (Table I). Its logical data connections are explicitly established by data coupling arrows. In addition, the functional capability represented by the process boxes may be decomposed in subordinate diagrams until an acceptable level of detail is reached. Finally, the notation is rich enough to represent loops, decision branches, recursion, and the existence of pre-built modules.

3) Structure charts have the capability to represent near-implementation design. While this is not necessary during the preliminary design phase, it is a desirable characteristic because it allows the preliminary design and the detailed design to be represented by the same graphical technique. This consistency during the design phase can increase the savings in time, effort, and design reliability.

TABLE I  
STRUCTURE CHART GRAPHICAL SYMBOLS

NAME	SYMBOL	EXAMPLE OF USE
PROCESS		
VECTOR		
DATA (SHADED FOR CONTROL)	 	

### The Preliminary Design

The complete Preliminary Design is the textual and graphical model presented in Appendix B. At the top-most level the Preliminary Design mirrors the information which was presented in the Functional Model. Indeed, the majority of the Preliminary Design reflects much of the same information as the Functional Model. The differences between the two are their purposes. The Functional Model defines the requirements by identifying the necessary functions in the Test Methodology. It also gives initial insight to the Methodology's data flow. The Preliminary Design diagrams give an explicit show of the process hierarchy. Additionally, the data flow, while still a logical representation, is more closely associated with the true system data flow (e.g. process and control data are explicitly shown).

Through the process of validation between these models, they have come to appear quite similar. As the Functional Model partitions the Methodology into the lifecycle phases, so also does the Preliminary Design (figure 8). However, the Preliminary Design includes new information, because it presents the system in greater detail. An example of this is the diagram for node 1.1.3 (figure 9), which defines the subprocess for node 1.1.3, and explicitly shows the difference between process and control data. As a logical

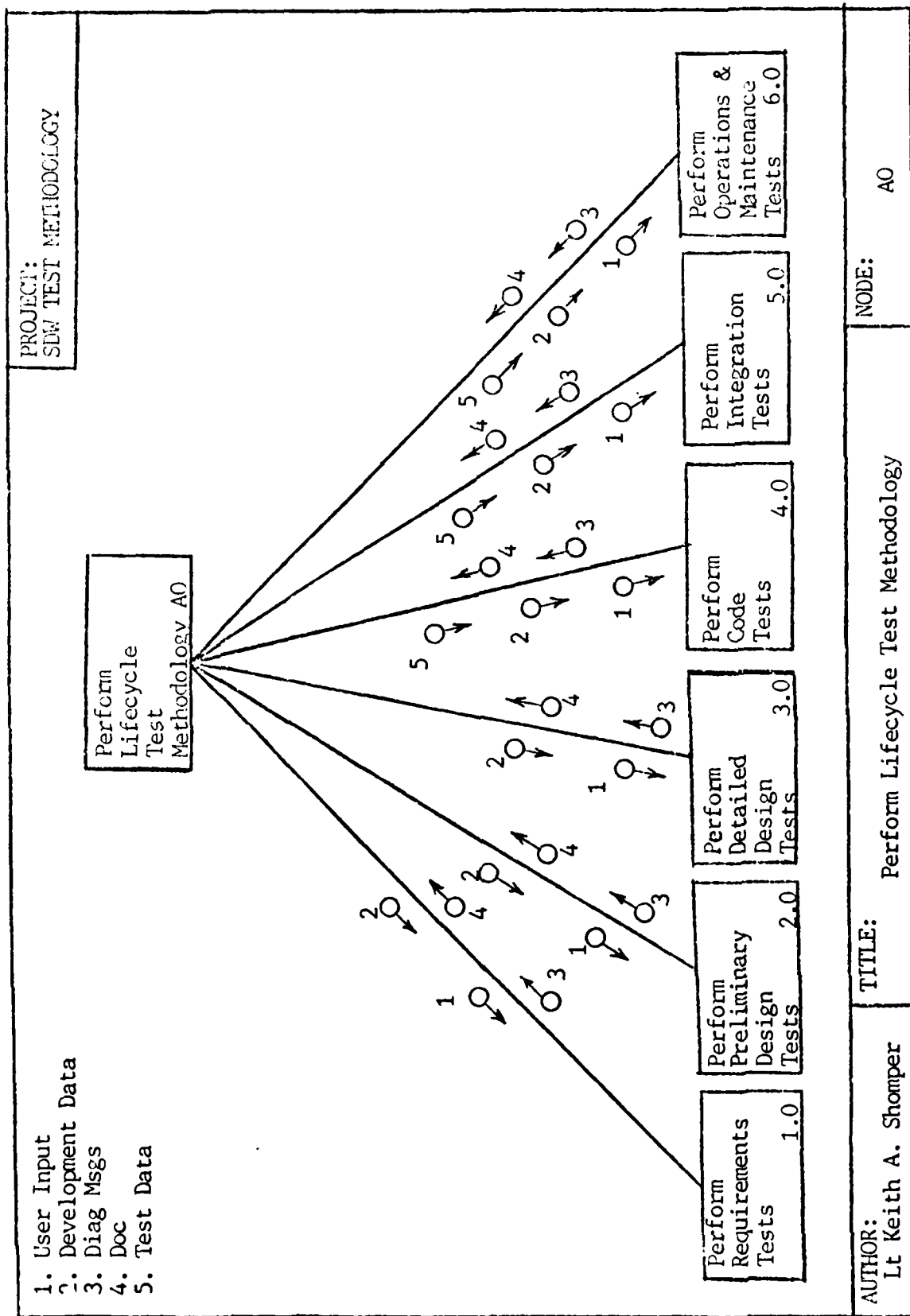


Figure 8: Preliminary Design A0 Diagram

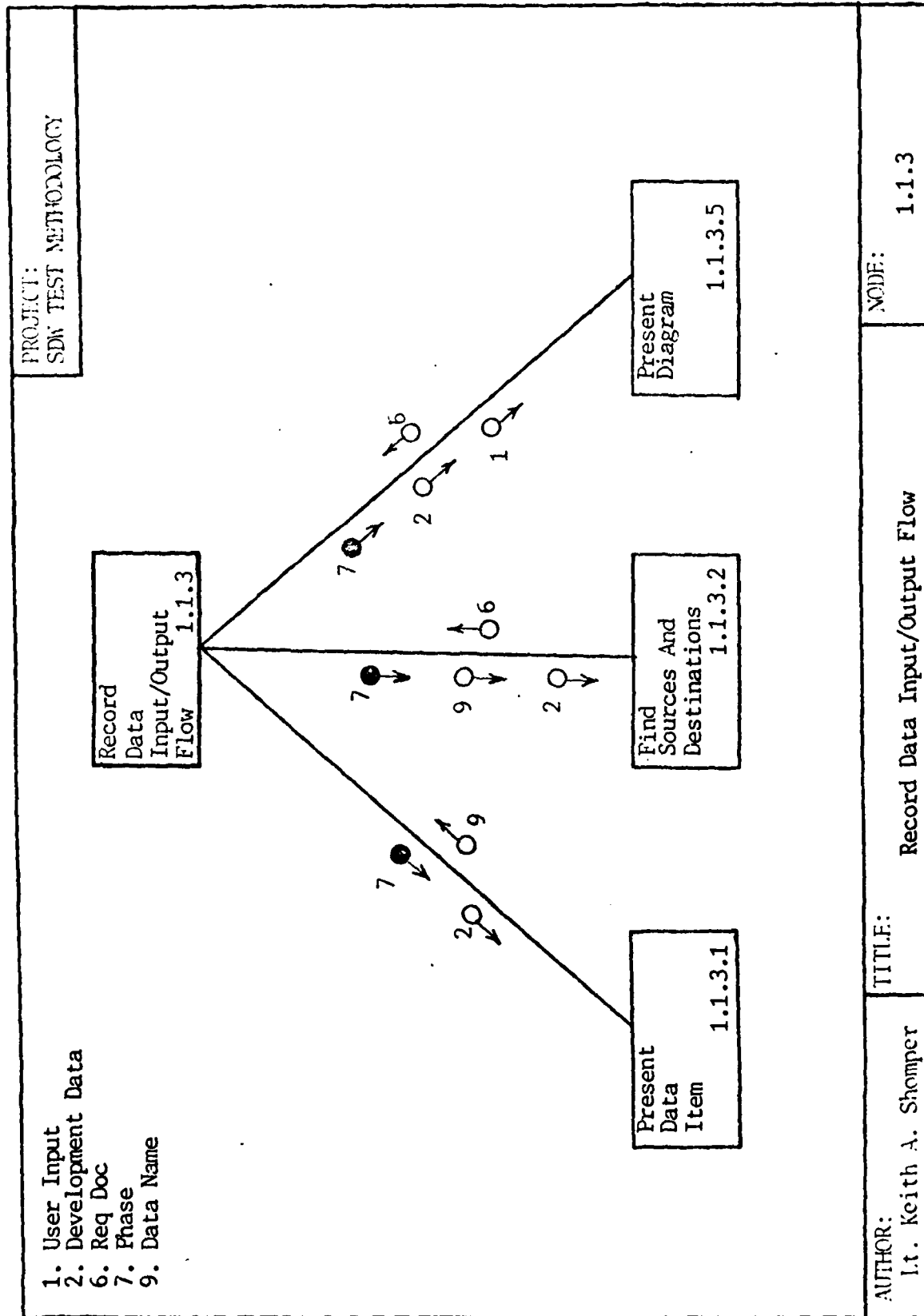


Figure 9: Preliminary Design Node 1.1.3

design representation the Preliminary Design must undergo further refinement before it can be transformed into code. The refinement of the Preliminary Design occurs in the next chapter, the record for the detailed design phase.

#### Summary

The preliminary design phase is the high-level design phase which allows for the transition from the abstractness of the requirements to the detail in the detailed design. This chapter is a record of the preliminary design phase for the SDW Test Methodology. It gives support for the Preliminary Design in four ways, by outlining specific concerns to the Preliminary Design in the AFIT environment, by providing a Configuration Model to support of the Test Methodology software, with textual justification and resolution of the requirements in Chapter Two, and with the textual and graphical information presented within Appendix B.

As the design begins the software engineer must begin to consider various attributes that may affect the integrity of the design. These attributes are generality of design, budget, technology, capabilities, and environmental constraints. Attention to these things help the software engineer focus on how the requirements can be met in a realistic environment.

The Configuration Model is presented to outline the system support (the hardware and software) for the Test Methodology. This Model also explains how the Test Methodology interfaces with the SDW as it is currently configured. The Configuration Model is presented in Appendix E.

Textual justification for the Preliminary Design is presented in the section on the resolution of requirements. This section describes how the Design and the Configuration Model support the requirements listed in Chapter Two by addressing each requirement separately.

The Preliminary Design is presented in Appendix B. Included with the Preliminary Design is information describing the differences between the Preliminary Design and the Requirements Model in Appendix A. Structure charts were chosen for design representation because of their explicitness in presenting the design at this level and their understandability.

The information presented in this chapter and its related appendices is the baseline for the design of the SDW Test Methodology. The choice to use structure charts as the design representation becomes important as the development enters the detailed design phase, because of the flexibility of structure charts to represent both high-level and detailed design. This investigation begins at this



point to narrow its focus concentrating on particular portions of the Test Methodology. This is done in order to implement those parts of the Methodology that have been designated as immediately useful to AFIT software development in the time allotted for this investigation. Presented below is a prioritized list which designates the order in which modules are added to the Test Methodology.

- 1) The Control and Menu Structure
- 2) The Consistency Tests
- 3) The Correctness Tests
- 4) The Clearness Tests
- 5) Reconfiguration of the SDW's Resident  
Diagnostic Tests
- 6) The Tracability and Correctness Tests
- 7) The Execution Test/Rehosting of a PDL  
Processor

## Detailed Design

### Introduction

Detailed design is the third phase in the software lifecycle; in this phase the physical design of the software is produced. Production of the physical design, or detailed design, begins once the preliminary design is nearly complete. To transform the logical design into the physical design the logical, or preliminary, design is successively decomposed into smaller segments until the problem solution can be modeled in a physical design.

The purpose of the detailed design phase is to allow for transition between the abstract logical design structure developed in the preliminary design phase and the implemented solution. Because the detailed design spans the gap between the graphical representations typical in preliminary design models and the code of the implementation phase, the detailed design often includes elements of both representations, graphical and lexical. The alternatives for these representations are discussed later in this chapter.

There are three main sections in this chapter. The first is an overview of design issues that present themselves as the physical design is addressed. The second section provides background material on a few design methods

which are pertinent during the detailed design phase. The second section concludes with the selection of a design method for the SDW Test Methodology's Detailed Design. The third section introduces both the Structural Design, the graphical representation of the Detailed Design and the Algorithmic Design. These designs are presented together in Appendix C of this investigation. The Algorithmic Design, the Structural Design, and this chapter combined comprise the record for the detailed design phase of the SDW Test Methodology.

#### Detailed Design Issues

In Chapter Three five concerns were presented which the software engineer must address during the preliminary design phase. These concerns dealt with high-level development issues (e.g. budget and resources); therefore, they drew interest to themselves during the initial design phase. Similarly, the latter design phase, the detailed design phase, has its own unique issues which must be considered. The following list presents these issues (32, 37):

- what support programs and algorithms are available,

- what data structures are necessary, and

- the interface compability of the software system to its environment.

The order of the list does not indicate a mandatory order in which these issues must be considered. Indeed, the data structures might be defined prior to anything else to help the software engineer to understand his problem (1), or the interfaces might be pre-defined by existing standards or associated software. Whatever the situation, these issues must be addressed during the detailed design phase.

The first issue concerns itself with reducing the problem domain. The essence of this task ought to have been exercised in all of the prior development phases; however, the focus is tighter in the detailed design phase concentrating on available applications software (e.g. database managers) and supporting algorithms (e.g. sorting algorithms). The purpose of these software searches is to keep from "reinventing the wheel" when programs already exist to perform a specified task or when the task may be performed by a combination of existing programs (26, 27). The software engineer might also search for algorithms to help structure the problem and initiate a more efficient and effective design with less difficulty than a completely original solution. Discerning the point at which the original development can be phased out in favor of existing programs and algorithms relies on the individual software engineer's experience and his knowledge of what is available in commercial, in house, and public domain software.

The second issue addresses the development software's data structures. In the preliminary design phase only a conceptual understanding of the data flow was necessary to understand the design adequately. As the design becomes more detailed, the structure of particular data items must be defined. Additionally, decisions concerning the methods of data storage and retrieval become important. In some environments it is necessary to construct abstract data types (ADTs) to realize particular high-level functions in an easy to understand manner. An abstract data type is an ordered triple consisting of the data types' domains, the functions which operate on those domains, and axioms which define the functions (32:7). Visualizing the data flow under the concept of ADTs helps "clean up" the design by hiding the "messy" implementational details; this concept is known as information hiding.

If the files are considered as macro data items, then they may be defined in much the same way as the data structures. Because files are physically and conceptually larger program elements, their definition can proceed the detailed design; however, if they are not significantly complex, then they are presented with the detailed design. The files which are used in the Test Methodology are discussed later in this chapter when the issues on the selected DBMS are presented.

The third and final issue addresses the software's interfaces to its environment. In particular, the software engineer must begin to visualize how the other hardware and software items will couple to the developing system once it is implemented. The interface issue is addressed during the detailed design phase to insure that the software's compatibility with its related systems is not accidental or an afterthought.

There are a number of important considerations for the software engineer within this topic of interfacing; these considerations are listed below:

- communication protocols, initiating or terminating

- special device setups (e.g. terminal setup) before establishing software links

- special syntax required for imbedded program operation

- specific port addresses or hardware registers to access or initialize

- message transfer timing, timeouts, and other timing considerations

- data sequence or parameter order

- interfaces with incomplete or non-permanent devices or code

- interfaces with concurrent or future development.

The first six considerations mainly address concerns with coupling to existing hardware or software. In this case designing the complementary interface is somewhat simplified, because a defined standard for that interface already exists (at least in part). The last two considerations present a more complicated problem, because the interfaces must be defined and agreed upon by all the software/hardware development groups involved or by a managing authority. Complications which may arise are misunderstandings and interpretations about proposed designs, changes in the interface design during the design phase, or lack of management between the development groups with regard to compatibility issues.

These three main issues; program structure and algorithms, data structures, and software interfaces, must be examined during the detailed design phase. How these issues are confronted in the Detailed Design of the SDW Test Methodology is presented in the following sub-section.

Detailed Design Issues for the SDW Test Methodology. Three major issues are considered during the detailed design phase of this investigation which had a significant impact on the Detailed Design of the SDW Test Methodology. These issues are discussed here with specific examples relating to the more general topics preceeding this sub-section.

Software Tools and Algorithms. The first issue addresses software tools and algorithms which aided the author in the detailed design phase. In the initial phase of design, the preliminary design, it became evident that many testing functions, particularly those applicable to the implementation, integration, and maintenance phases, could be supported by existing software tools in the same manner that the SDW is supported (25:96). These tools were selected in the detailed design phase; they are cited here:

compilers

- VMS PASCAL compiler
- VMS FORTRAN compiler
- VMS BASIC compiler
- VMS COBOL compiler

linkers

- VMS linker facility

text editors

- VMS EDT editor

comparators

- VMS difference facility

debuggers

- VMS debugger

By using existing diagnostic tools to implement the majority of the final three lifecycle phases, the design for these phases is reduced to control and integration of the software tools and a few relatively simple interactive procedures.

Data access is another area in which existing software affects the Test Methodology's Detailed Design. The SDW Test Methodology is initially designed to interface with the



INGRES database management system. The INGRES system was chosen because of the simplicity of its manipulation language and its availability at AFIT. TOTAL was the only other DBMS considered as an alternative to INGRES. It was not selected because it was evaluated as less friendly with respect to its manipulation language in a direct comparison of the two systems. INGRES affects the Detailed Design's data access modules, because they are designed to take advantage of the data accessing capabilities of relational databases. The database relations and their attributes also affected the Methodology's design by varying degrees according to their definition. These definitions were identified by students and faculty at AFIT with regard to this and related development efforts.

Finally, as the Test Methodology's design is successively decomposed in the detailed design, it is apparent that certain functions can be supported by existing procedures and algorithms. One such example is the treeprint and tree traversal algorithms used in parts of the requirements and design "correctness" tests. These algorithms are modified versions of similar algorithms developed and studied by AFIT students in algorithm design and analysis courses. Additionally, a procedure selection routine similar to that used in the Computer Aided Design Package (46) and in the ICECAP (64) system is employed for

selecting the test modules in the Test Methodology.

Data Structures. Another issue which became apparent during the detailed design phase was that the data and file structures which the Test Methodology would use needed to be identified. Two types of data structures are identified as necessary; they are trees and lists. The file structures fall into three categories, INGRES manipulated files, text files, or software generated files.

The tree data structure is used in a number of places to support algorithms testing "correctness", consistency, and completeness criteria on the requirements and design graphical models. The tree structure is an appropriate choice, because the SADT and DFD models can be mapped into it by a one-to-one mapping of activity boxes or bubbles to tree nodes. Tests may then be executed using tree traversal algorithms to "visit" each node and perform the appropriate tests. Structure charts also lend themselves to a similar type of mapping; therefore, they can be tested in the same manner.

The second application of the tree structure in the Test Methodology is the menu structure. The menu represents the control structure which was provided for in the Preliminary Design. Representation by a tree structure is an appropriate choice, because of the menu's hierarchical format. The menu structure designed for this investigation

is an adaptation of the menu data structure created in (46) and also (44); this type of structure was chosen because of its modularity and modifiability. Alternative menu structures such as that employed by Hadfield (25) and those typical in interactive programming (21) were considered, but they were rejected, because they lacked modularity.

Lists are the second type of data structures that are used throughout the Detailed Design. The primary reason for using lists is the ease by which they are manipulated by INGRES when they can be represented in the INGRES QUEL language. A secondary reason is that lists serve well as rosters providing development information while controlling iteration in many of the algorithms. It is noted here that the data structures used in the Test Methodology are not limited to lists and trees, but that these two are presented because they are the most commonly used herein. As mentioned earlier data structures (or abstract data types) allow the software engineer to discuss the problem on "high-level" terms. The Test Methodology attempts to support the practice of using ADTs through its use of trees, lists and other necessary data structures.

The file structures that are necessary in the design of the Test Methodology are divided into three classes, INGRES manipulated files, text files, and software generated files.

The structure of the INGRES manipulated files are not pertinent to this investigation, because they are only accessed through INGRES. The text files are pre-defined files containing information necessary for the menu discription, the help facility, and dictionaries. They are also generated for documentation purposes during various testing functions. Finally, the Test Methodology operates on such files as are generated by the pre-built diagnostic tests (e.g. complier). These files are mainly restricted to data, source, object, and command files.

Interface Issues. The third issue which becomes important during the detailed design phase concerns the software interfaces. The SDW Test methodology is not a stand alone system, but is intended to support testing in an automated interactive software development environment. The environment for the initial implementation is the Software Development Workbench; the interface to the SDW is the first type of interfacing problem the Test Methodology's design must solve: interfacing with an existing system. The Methodology's interface to the SDW is designed according to a pre-defined outline (25:358). Those additional interfaces to the SDW which are necessary for the Test Methodology to access particular SDW component tools are provided for in the Test Methodology's menu selection routine.

The second type of interfacing problem which the

Detailed Design must solve also involves interfacing with an existing system; this interface is with the INGRES database management system. This interface problem has two domains, the first is to master the QUEL language in order to use INGRES, and the second is to coordinate with other concurrent investigations to define the relations and attributes in the database. These problem domains must be solved in the detailed design phase before development can continue in this area to insure that all the necessary development information can be accessed and tested. In addition to solving these problems this investigation is keeping the Test Methodology as independent of INGRES as possible by modularizing every access to INGRES.

The third type of interfacing problem is that one which occurs when interfacing between two developing subsystems, in this case the Test Methodology and the Data Dictionary Generation Tool (58). This problem also has two domains, a software interface domain and a compatibility management domain. The software interface is developed within the interests of both projects to include giving a Test Methodology user the capability to store information in the development database through the Data Dictionary Generation Tool. To achieve true compatibility between systems they must be designed with a general idea of the each other's purposes and each other's goals; this is the compatibility

management domain. The issues that were addressed between the two investigations in this domain are too lengthy to be discussed here, but a short list is provided to identify some of these questions and concerns which did arise:

what information was necessary to carry in the database,

how should the information be presented, by what relationships,

what was each software engineer's conception of the end user,

how the two systems would interface, and then how accessible should they be to each other's information,

additional topics on newly discovered requirements which the integrated system should satisfy and how these on-going requirements would be satisfied.

It is evident that when tools are design to go beyond interfacing, to the degree in which they support each other; that they must also be designed for compatibility beyond their physical interface. The Test Methodology attempts to do this with all its interfaces.

#### Methods for Creating a Detailed Design

The purpose of the Detailed Design is to provide for a transition between the Preliminary Design and the program code. It begins with a refinement of the Preliminary Design's low-level functions and it ends when the succeeding design may be directly implemented in the intended programming language. There have been numerous methods

suggested for use in detailed design (10, 22, 23, 26 40, 47, 65), but only the methods which are considered for the Test Methodology's Detailed Design are discussed here.

Evaluation Criteria. Before the representational choices for the Detailed Design can be considered a set of evaluation criterion must be established. With this criteria established, a number of the design methods may be immediately removed from further consideration in this investigation. Those methods which remain are then evaluated in greater detail and a specific method is chosen. The following is a list of the criteria and a brief description describing the meaning of each:

Availability - Tutorial information must be available on the use of the method, and if automated support is required, it must also be available.

Aid to Development - The method's notation must support the commitments made in the software's development during the Preliminary Design without extensive re-design. In other words, the detailed design representation must pick up where the Preliminary Design left off, without major re-design or re-invention.

Information Completeness - The detailed design method must be able to capture all the significant design information in the Preliminary Design to insure design integrity.

Ease of Use - The method must be simple to use and not require much training to become a proficient user

Testable - The method must be lend itself to testability or provide its own diagnostic tools for testing.


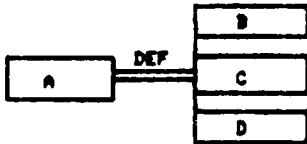



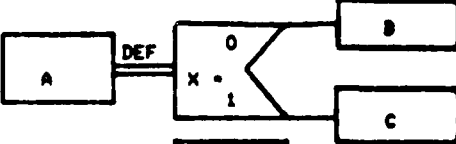

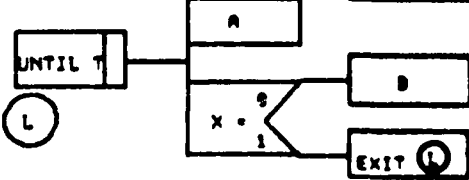

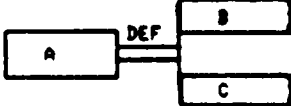
These criteria are determined after considering the time which is available to the student software engineer in the AFIT environment for software development, in particular detailed design. They also reflect an attempt to use a detailed design method which the Test Methodology can support. Finally, they require that the method's notation be sufficiently powerful to carry the design from the preliminary design phase to implementation phase without a loss of design information; this capability promotes top-down design. Six methods satisfied these criteria in representing the detailed design and are evaluated below by identifying their individual strenghts and weaknesses. Once these characteristics are identified they are compared with each other and a "best" representation for the Detailed Design is chosen.

The Methods. The detailed design methods selected for evaluation are Problem Analysis Diagrams (PAD), structure charts, flowcharts, Program Design Language (PDL), META stepwise refinement, and pseudocode. These six methods are divided into two sub-classes for evaluation. The first sub-class contains the methods with a graphical design representation; these are the first three methods in the above list. The latter three methods use a lexical design representation to present a detailed design.



Graphical Design Methods. Three graphical techniques are evaluated in this section; they are PAD, structure charts, and flowcharts. PAD was developed as an improvement of Warnier diagrams (22). The PAD method uses five symbols in its representation in addition to descriptive text to present a detailed design (Table II). Using the five symbols PAD is capable of supporting the whole range of design levels from high-level preliminary designs to near-implementation detailed designs. This capability is realized through the PAD's promotion of top-down design techniques and the def symbol. PAD's


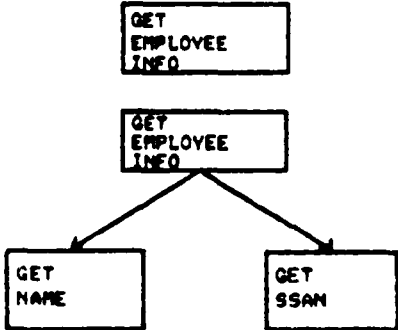


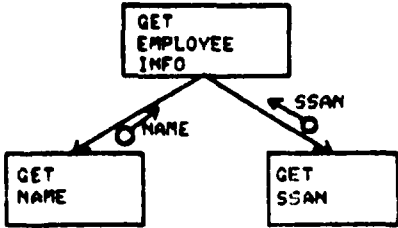
TABLE II PAD GRAPHICAL SYMBOLS

NAME	SYMBOL	EXAMPLE OF USE
PROCESS		
REPETITION		
SELECTION		
LABEL		
DEF		

strengths are its explicit show of control flow and nesting levels, its ability to represent parallel processing, and its automated tool support. However, PAD designs do not make a distinction between the logical and physical levels of design, neither do they show data flow, which reduces their effectiveness as a system-level design technique.

The second technique for evaluation is structure charts. This method of design has been in use since the late 1960's (60), hence it is understood by most software engineers. It has three basic constructs to represent a design (Table III); some "versions" of the structure chart



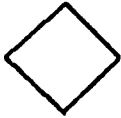
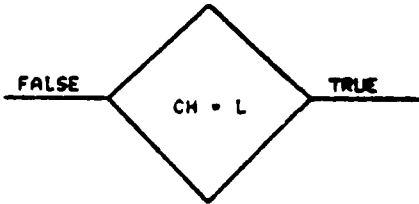

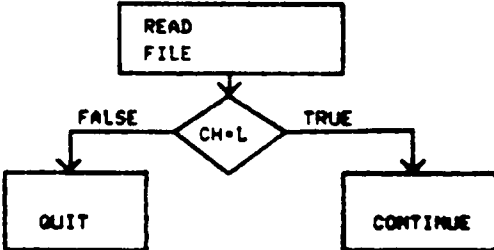
TABLE III  
STRUCTURE CHART GRAPHICAL SYMBOLS

NAME	SYMBOL	EXAMPLE OF USE
PROCESS		
VECTOR		
DATA (SHADED FOR CONTROL)		

technique include additional symbols to show iteration, selection, reused modules, or pre-built modules. Like PAD, structure charts are capable of representing the full range of design levels, albeit less capable than both PAD and flowcharting in showing control flow at the near-implementation level. The strengths of structure chart design is its explicit show of hierarchy, decomposition, and data flow, as well as its use and applicability in promoting clean module interfaces. Its greatest weakness is its stand-alone capability for representing a detailed design; generally structure charts need to be supported by pseudocode at the detailed design level.

The third technique evaluated is the flowchart. Used since the 1940's, originally as a way to document programs then later as a method of design, the flowchart has evolved into a number of different design techniques (48:93). The particular flowchart representation evaluated here is the ANSI flowchart (10). ANSI flowcharts use three graphical symbols and text to illustrate their designs (Table IV). In some detailed flowcharts the text will resemble pseudocode. While the flowchart is not particularly useful in high-level design, it can be used as a follow-on technique for low-level modules. It excels in showing control flow which can simplify module testing, but it suffers from a lack of

TABLE IV ANSI FLOWCHART GRAPHICAL SYMBOLS

NAME	SYMBOL	EXAMPLE OF USE
PROCESS		
DECISION		
CONTROL FLOW		

structure and non-adherence to the principles of top-down design.

Comparing of the above methods to select a single technique shows that flowcharting is too unstructured and at times too detailed to satisfy the Test Methodology's needs for detailed design. At the outset PAD seemed to be the ideal representation; however, it is rejected, because it lacks the notation to represent data flow and system design characteristics which are necessary to make the Test Methodology a modular flexible system. Therefore, structure charts are chosen as the "best" representation. In addition

to the benefits which have already been cited, they permit the investigation to continue the design with the same method used in the Preliminary Design; this is an aid when the time factor for the design is considered. As mentioned above, structure charts are not usually sufficient to stand alone in the detailed design; therefore, an additional representation is chosen from one of the following lexical design methods.

Lexical Design Methods. Three methods are chosen as candidates to support the structure chart technique they are pseudocode, Program Design Language (PDL), and META stepwise refinement. PDL is similar to pseudocode and at times is classified as such; therefore, the distinction between PDL and pseudocode is clarified here at the outset. The distinction this investigation makes between the two is that PDL is more formally defined and can be identified as one particular language. Pseudocode is an individually defined language; it has no rules for structure or content, but it is tailored by each individual user to meet that particular software engineer's tastes and style of programming.

Pseudocode is the first candidate considered for evaluation. Because pseudocode easily captures important elements of low-level design, without restrictions to

TABLE U  
COMMON PSEUDOCODE CHARACTERISTICS

CHARACTERISTIC	COMMON CONSTRUCT	EXAMPLE USE
DELIMITERS	BEGIN END	BEGIN X = X + 1 Y = Y + 1 END
LOOPING CONSTRUCTS	DO WHILE REPEAT UNTIL ENDDO	DO WHILE X < 20 X = X + 1 Y = Y + 1 ENDDO
DECISION CONSTRUCTS	IF THEN ELSE IF ELSEIF	IF X = 0 THEN X = 1 ELSE X = 2
NATURAL LANGUAGE	—	IF NO MORE DATA ITEMS THEN GET MORE ITEMS FROM LIST
NESTING LEVELS	—	DO WHILE IF ... THEN REPEAT X = X + 1 UNTIL X = 20 ELSE DO WHILE ...
INTERFACE PARAMETER LIST	PROCEDURE NAME(VARLIST)	GETINFO(NAME, NUMBER)

implementational syntax, it is the method most often used to support structure charts (Table V). Other strengths of pseudocode are its flexibility and its heuristic appeal as a top-down design method. Although, pseudocode is generally tailored to each individual software engineer's style, there are characteristics which are common to "good" pseudocode designs. Even "good" pseudocode designs have their drawbacks the majority of which result from the lack of formalism in pseudocode syntax and semantics. The free format and semantics encourage misunderstandings in a design team over meaning, inhibit automated testing, and generally

AD-A152 857

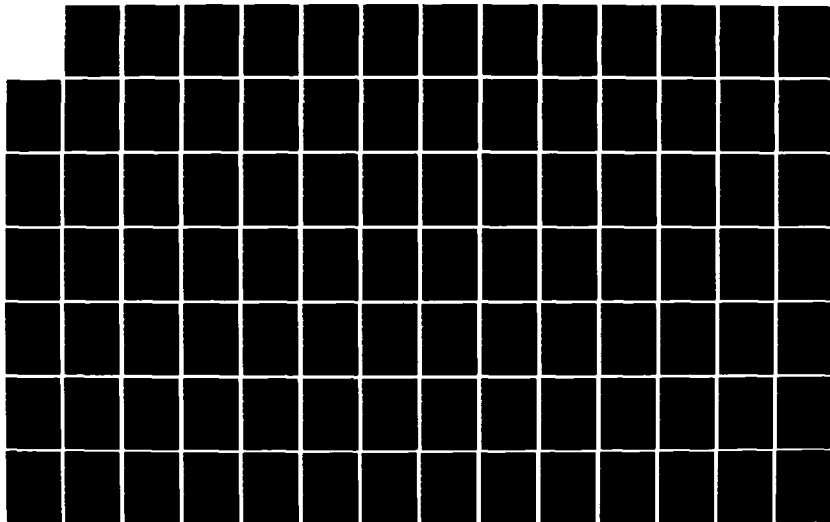
A TEST METHODOLOGY FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT(U) AIR FORCE INST OF  
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.  
K A SHOMPER DEC 84 AFIT/GCS/ENG/84D-26

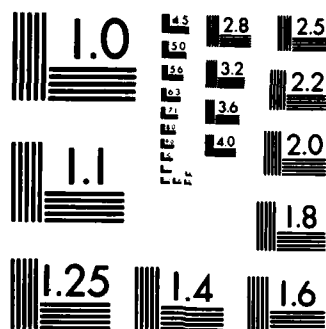
214

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A



aid only the "good" software engineers in producing "good" pseudocode. Because it resembles high-level programming languages, pseudocode can also lead to pre-mature coding.

Program Design Language (PDL) has many of the same characteristics of pseudocode (Table VI); it may, in some ways, be thought of as structured or formalized pseudocode. PDL was developed to support the creation of structured designs through top-down design techniques. The formalization of PDL is a result of adhering to specific control constructs and keywords. The strengths of the PDL method are its capability to present the design in a natural language, its ease of use, its promotion of structured

TABLE VI PDL CONTROL CONSTRUCTS

CONSTRUCT	EXAMPLE USE	CONSTRUCT	EXAMPLE USE
DO	DO WHILE MORE DATA READ DATA PROCESS DATA WRITE DATA ENDDO	UNDO	DO WHILE MORE DATA READ DATA IF DATA IS STOP UNDO-- PROCESS DATA WRITE DATA ENDDO WRITE NAME <----- (CONTROL PASSED TO FIRST STATEMENT AFTER DO
IF	IF MONDAY WRITE THESIS TYPE THESIS ELSEIF TUESDAY BUILD SADT'S DRAW DFD'S ELSE PLAY GOLF ENDIF	CYCLE	DO WHILE MORE DATA <----- READ DATA IF DATA IS OVER CYCLE-- PROCESS DATA WRITE DATA ENDDO
		DO CASE	DO CASE OF OPERATION ADD: ADD RECORD DELETE: DELETE RECORD RECORD NEW STATUS OTHER: ISSUE ERROR MESSAGE ENDDO

control flow, and its attention to module and system level interfaces and to error condition responses. In addition, PDL is supported by a computerized processor to help the software engineer evaluate his design. A weakness in PDL is that it can lead to pre-mature coding which can result in software structural errors.

The final method selected for evaluation is META stepwise refinement. META stepwise refinement is mainly the rules for effective software design formalized into a procedural design method. The distinction between META stepwise refinement and top-down design is subtle. Top-down design is progressive refinement on a single design through an iterative process. META stepwise refinement employs the iterative problem solving technique, but proposes n more detailed solutions and a selection of the "best" of these solutions for continuance as its iterative process (figure 10). The value of the META method is its procedural approach to detailed design. However, it lacks guidelines and standards for the software engineer in selecting the solution which is "best". Also, because procedural formalism can be promoted in most any design method, META stepwise refinement is not truly unique.

These are the three methods evaluated for support of the structure chart technique. A comparison of these methods reveals that PDL is the best method in this

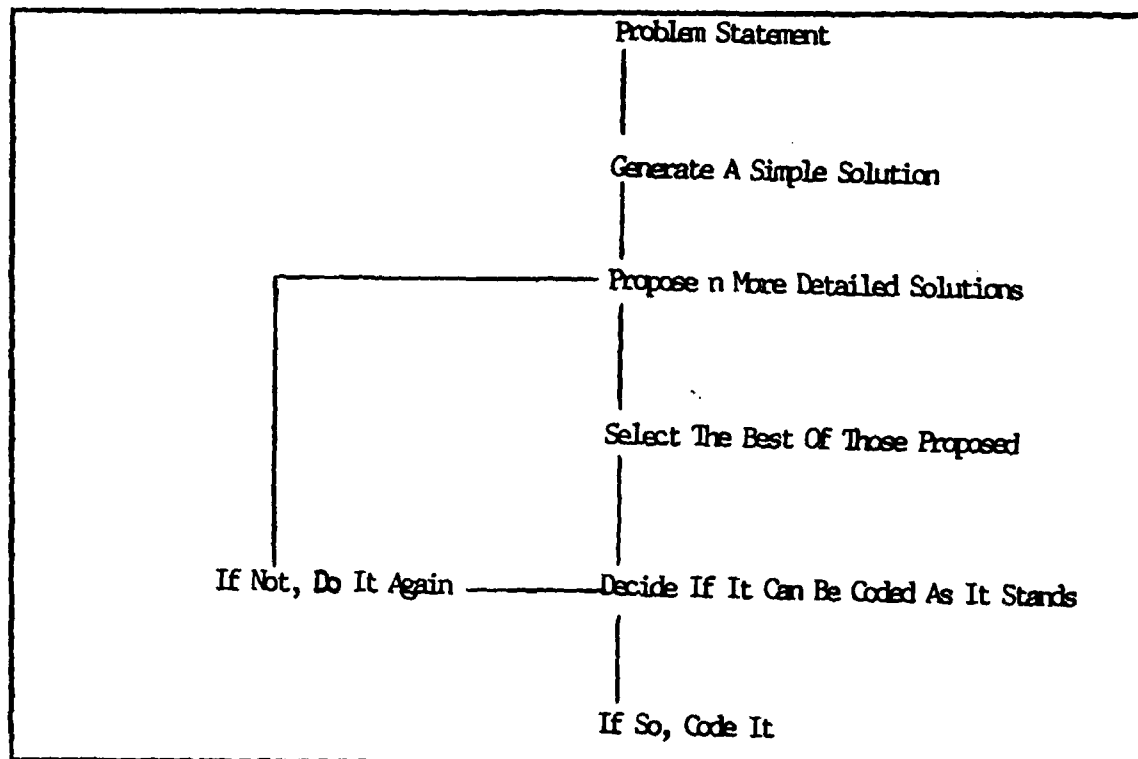


Figure 10: Meta Stepwise Refinement as an Iterative Process  
Source: Software Design Methods and Techniques

investigation's application. The choice is based on the weaknesses possessed in the pseudocode method because of its free format and semantics; weaknesses which are resolved somewhat by the formalism of PDL. META stepwise refinement promoted effective software design, but lacked any truly unique ideas which could not be incorporated into the PDL method. Additional reasons for choosing PDL are its compatibility with the data dictionary method of representing software; moreover, although pseudocode is also compatible with the data dictionary, PDL is more formal and hence more readily testable than pseudocode.

### Presentation of the Detailed Design.

With the methods of representation for the Detailed Design determined, the focus of this chapter turns to the Structural Design and the Algorithmic Design. These two designs are presented together in Appendix C. The format of presentation is to first display a Structural Design structure chart followed on the next page by that chart's cooresponding algorithm in the Algorithm Design. This format is chosen for convienence, practibility, and understanding. The convienence and practibility of this format is revealed when the design is evaluated. During the evaluation a reader may quickly grasp the functional intent and the software structure of the design by examining the structure charts. Then if more detail is required, that portion of the Algorithmic Design is readily available to the reader for review. Condensing both designs together promotes understandability because together the illustrate all facets of the design. Moreover, they are placed together because they represent one design, but they are seperated on different pages so that individually the Structural Design can be compared to the Preliminary Design on issues addressing software structure, and the Algorithmic Design can be compared to the software code on issues addressing contol and data flow. The Detailed Design is

unique in this type of representation, because it spans the gap between the software's graphical models and its program code.

#### The Detailed Design

The Detailed Design is the combination of the Structural Design and the Algorithmic Design. Because it represents the system as closely as possible, that is without actually being code, it can appear quite different from a high-level preliminary design. This is the case with the Test Methodology's Detailed Design. Because the Preliminary Design reflects only the structure, calling order, and logical data flow of the Test Methodology, it more closely resembles the Functional Model. However, the Detailed Design has now included the logical hierarchy of the top three levels into its menu structure (figure 11). The tree structure in the top of figure 11 is a scaled down representation of the preliminary structure chart design. The tree represents the top two levels (or diagrams A0 and 1.0 - 6.0) of the preliminary design. Each of these logical functions are supported through the menu SELECT routine and the menu structure. This structure is represented by the second tree, diagrams A2 and A26. The arrows are representative of interfaces for the individual tests. That OFFER\_MENU has six support routines (as many as there are lifecycle phases) is nearly coincidental.

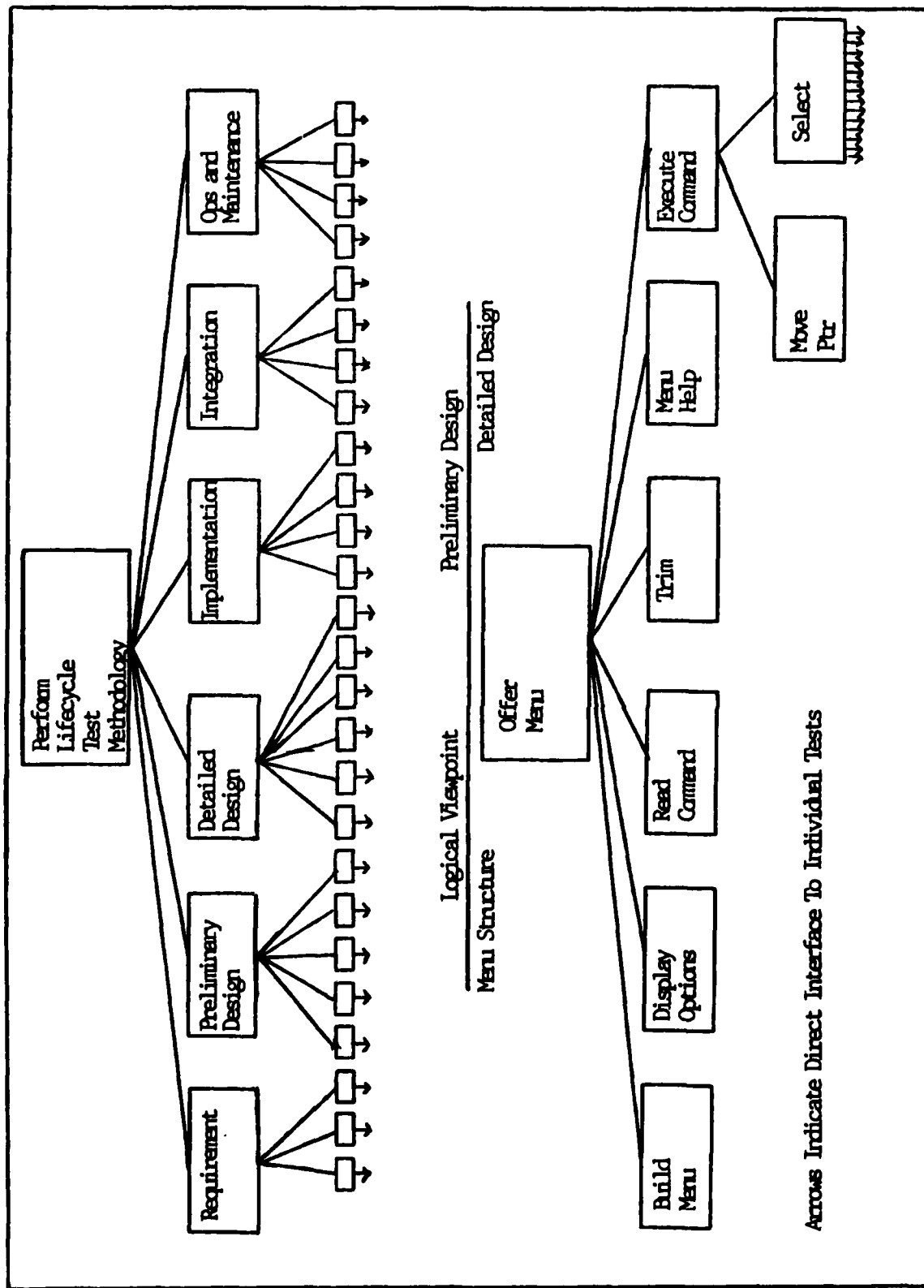


Figure 11: Conversion of Logical Structure to Physical Design Structure

The combination of the top levels into the menu is possible and practical because the functions at these levels simply pass control to the next lowest level of functions. Once this case is realized, a design is created which represents the necessary menu structure; these are the design charts and pseudocode prefixed by a capital A in the node number. This numbering scheme is chosen to explicitly denote the modules which are associated with the top level logical structure of the Preliminary Design (this convention is continued into the implementation procedure numbering).

At the lowest level of the Preliminary Design are specific types of tests, yet with no detail on how they will be implemented. The Detailed Design also includes these tests (they are the ones which conventionally numbered), but includes part of the detail which the Preliminary Design ignored. Each of these test are designed independently of the others; this is possible if the interfaces to the menu are predefined. During the design of each test, it became apparent that a number of lower level modules are used by different tests, and that these modules are usually associated with data access. Therefore, all accesses for data are modularized to provide a library which each of the individual tests can use. The benefits of this decision are twofold. The first is that the Methodology's design is further modularized. Secondly, this permits the Methodology

to be designed independently of the database which will support it.

#### Summary.

The detailed design phase is the second phase of software design. In this phase the physical model, or detailed design, of the software is created. As each new phase is encountered, there are phase related issues which must be addressed. The issues addressed within this chapter are those that affected the composition of the Detailed Design in some way. Because the Detailed Design falls at the transition from representation by graphical models to representation in program code, it often contains characteristics of both representations, such is the case with the SDW Test Methodology's Detailed Design. This chapter identifies the representations selected for Detailed Design and justifies those selections; it also introduces the Detailed Design which is presented in Appendix C.



## Implementation

### Introduction

The implementation or coding phase is the forth phase in the software lifecycle. During this phase the detailed design is transformed from its model representation (the graphical constructs and pseudocode) to software routines or procedures in a particular implementation language. Implementation often begins before the detailed design is fully complete in order to test various aspects of the design for implementational feasibility before full-scale implementation begins. As an example, the "umbrella" structure of the software (e.g. the menu) may be implemented early to be evaluated for user-friendliness, robustness, and understandibility or to allow the proposed user to get a "feel" for the software before delivery. The latter case is typical of prototype software production. Although, the implementation phase may overlap the detailed design phase, care must be exercised in not implementing an incomplete design; the problems associated with such a case are presented in the previous chapter in the section on detailed design methods.

There are a number of items produced during the implementation phase. These items can be classified into two groups, the software related items and documentation.

The class of software related items includes the software, application databases, library support routines, and all other associated support files. In some cases this class can include specialized hardware support; however, this case is not addressed in the succeeding discussion due to its lack of applicability for this investigation. Documentation is defined to include all the items which are produced to help a system manager, software engineer, or user understand the purpose and the operation of the software or its related items. Before continuing a definition of software as it relates to this chapter is given. At the implementational level software can be represented in three forms, as the source code, the object code, or the executable image. Throughout the rest of this investigation, software and code refer to the source code unless otherwise noted. The source code are the computer instructions as they are written in the implementation language (e.g. FORTRAN).

#### Creating a "Good" Implementation

The software produced in the implementation phase is a direct result of the design preceding it, which is a direct result of the requirements definition. It is because of this relationship that lifecycle testing is effective. The purpose of lifecycle testing may be likened to a pipeline of sieves of successively finer mesh (figure 12): as the software's requirements, design and then the software itself

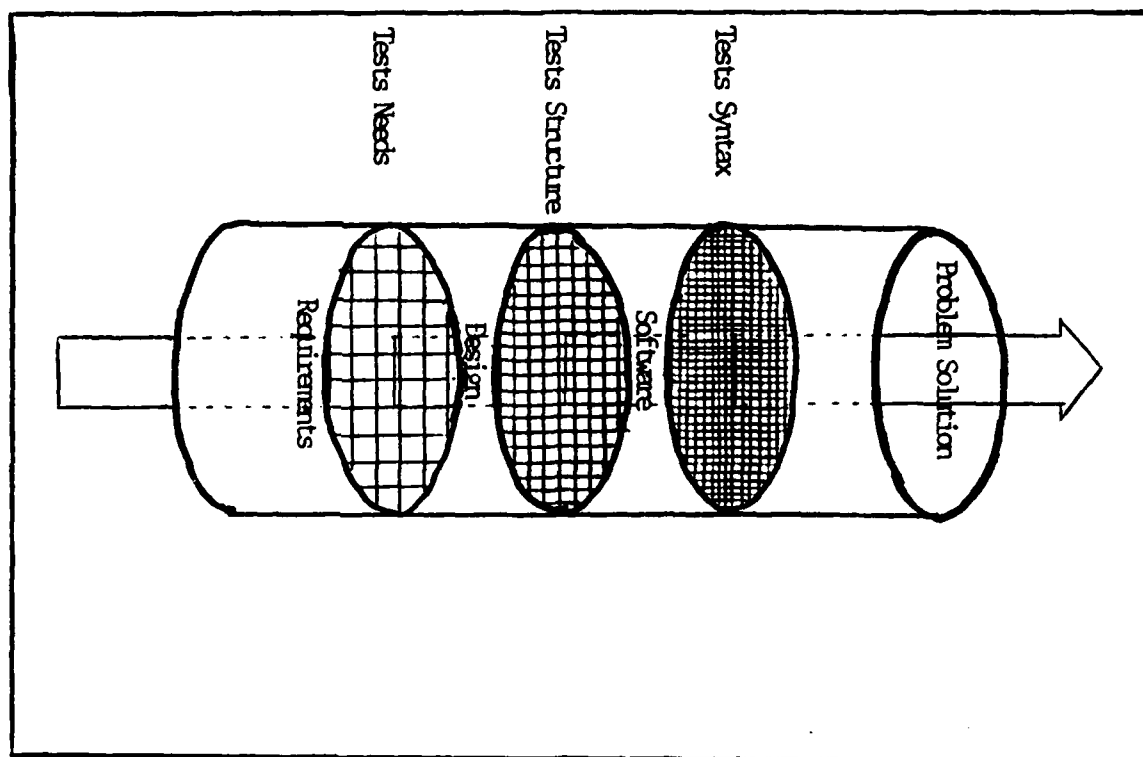


Figure 12: A Pipeline of Testing Sieves

pass through this pipeline, impurities (errors) are removed beginning with the "largest" (ill-defined needs) and ending with the "smallest" (syntax errors). In this manner, when the design phase is complete, the design will already exhibit most of the characteristics required in "good" software. Examples of these characteristics are modularity, defined parameter types, module data coupling, functional cohesion, and structured control logic. In the implementation phase the software engineer must select an application language which supports these characteristics, implement the design in that language, and apply the necessary run-time tests

(the finest sieves) to identify and eliminate the remaining errors.

The Implementation Language. As previously mentioned the software engineer's goal is to select an implementation language which supports the design characteristics and good programming style (36). In addition, a continuing goal is that the selected language support all the functional capabilities of the design. However, these two goals are not always mutually compatible, for example, a requirement for real-time response to user commands can lead to obscure but efficiently coded modules. The software engineer's responsibility is to evaluate the implementational tradeoffs to select to most appropriate language(s) from among the alternatives. This selection is also based on pre-defined evaluation criteria. Once the language is selected, the design is implemented in that language.

The Implementation Strategy. Suggesting that the language selection in the only process which preceeds the transformation of design into code is an oversimplification of the implementation phase. Prior to beginning the programming process, the software engineer should plan a strategy for implementation. This strategy is necessary to determine what the most efficient or practical order for coding the design modules. Factors which determine practicality are dependency (what depends on what to

execute), need, usefulness, and ease of implementation. Because the implementation phase can overlap with the detailed design phase, the implementation strategy may have already been partially addressed in the latter stages of the detailed design phase. However, before full-scale implementation begins a more detailed and refined strategy needs to be planned to govern the implementation as it progresses. The implementation strategy can also include a time-line of partial implementational goals to measure progress (55:151).

Implementation Testing. The basic idea behind implementation testing is to evaluate whether the given module executes in a prescribed manner. Various methods have been suggested to determine the outcome of the evaluation (34, 45, 67); they range from the rigor of program verification (67) to informal heuristic methods. Though these methods represent a wide range of program testing, each of them is simplified by applying the characteristics of structured design to the software and by choosing the "best" language available and an appropriate implementation strategy for the application. The implementation of the SDW Test Methodology is presented with this senario in mind.

## The SDW Test Methodology Implementation

The implementation of the Test Methodology began in the detailed design phase as it became apparent that the menu structure could be defined in a text file discription. By defining the menu this way, early implementation was possible, because it provided the menu structure with the flexibility to adjust to changes which could occur during the latter stages of the detailed design phase. In addition, the manipuation procedures for the abstract data types, were coded early due to a foreknowledge of INGRES data manipulation. Because of the overlap of the detailed design and the implementation, the implemantation language was selected prior to the composition of this chapter; however, the discussion on its selection, the criteria used, and the alternative languages are presented in this chapter, because this investigation's outline follows the more traditional lifecycle sequence of analysis, design, and then code.

The Language Selection. In selecting the implementation language for the Test Methodology three solution domains are identified. The intersection of these domains is the overall solution; these domains are the available languages, the user's needs, the selection criteria for the implementation language (figure 13).

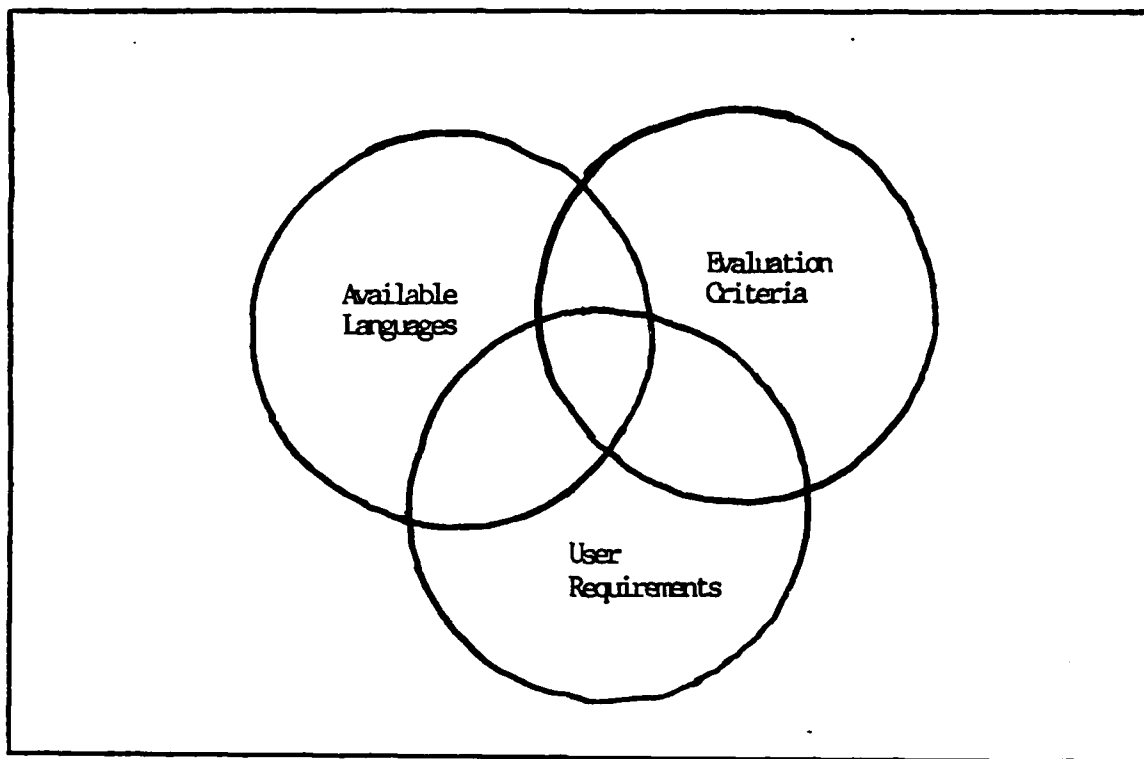


Figure 13: The Language Selection Domains

The Languages. Three languages are considered as candidates for the SDW Test Methodology implementation, they are FORTRAN, C, and Pascal. Digital Equipment Corporation's Command Language (DCL), COBOL, and BASIC are also available, however, COBOL is not considered an alternative, because this author has had no prior experience with COBOL, and it did not appear useful to learn for this application. BASIC is dismissed in favor of languages such as C and Pascal with which this author is more familiar. Finally, DCL is rejected because it lacks the capability to make the Test Methodology machine independent. The comparison of the remaining languages is discussed later after the evaluation criteria are presented.





Evaluation Criteria. The 1970's was a decade which saw a tremendous increase in the support for structured programming (35:31). There are many definitions for structured programming, but this investigation adopts the one supplied by Nickolus Wirth: "'structured programming' is the formulation of programs as hierarchical, nested structures of statements and objects of computation" (63). This definition emphasizes the necessity of modularity and logical control, characteristics which structured analysis and design can impart to the software implementation. Both of these characteristics, when supported in the implementation language, simplify the readability, testability, and maintainability of the software. In addition, they promote well-planned, well-considered program solutions. Other desirable characteristics which the implementation language should support are free format, natural language expressions, abstract data typing, and dynamic storage allocation. Each of these characteristics are presented below with the reasons for their inclusion in this list and their associated benefits. The order does not denote the degree of importance for these characteristics.

Modularity - modularity provides organization, a categorization of functional capability and process purpose. Modularization when combined with good coupling and cohesion provides for flexible "component" built software systems.

Its greatest contributions to the software are flexibility, understandability, testability, and maintainability (66).

Logical control - logical control refers to defined control structures, such as the DO in FORTRAN and the IF THEN ELSE in Pascal. Defined control structures (and the restricted use of GOTO's) promotes modularity by minimizing the number of logical entry points into a section of code. The result is increased understandability with respect to the control logic. Other benefits are better planned and formulated logic, and an increase in functional cohesiveness (35:41).

Free format - a language with a free format permits the software engineer to layout the written structure of the source code according to individual style. In other words, the software engineer may insert blank spaces or lines into the text of the source code along with comments to make the code visually pleasing. The benefits of free format are an increase in the readability and understandability of the program code (36:31, 66:212).

Natural language expressions - people do not think in machine language, and conversely computers cannot communicate in english. Therefore it is necessary to find a medium of communication between these two languages. A desirable characteristic is that this language be as similar to written english (or german, of french, ect.) as possible without introducing ambiguity. If this characteristic is present in the language, the software will be easier to read, to understand, to test, and to modify.

Abstract data types - ADT's are used for the same purpose as natural language expressions, that is to help the software engineer express the software problem solution in a familiar way. ADT's permit the software engineer to think in real world terms by collecting the data into logically defined objects. ADT's also improve testability by making the data flow easier to understand.

Dynamic storage allocation - dynamic storage allocation is useful in applications in which an undetermined amount of information must be passed from one module to another or stored by a particular module. Its greatest contribution is that it can save memory space if the amount of information varies widely from execution to execution.

The Selection. FORTRAN, Pascal, and C are the languages which are considered as alternatives for implementation of the Test Methodology. Each of the languages fulfill the user's needs domain, because each of the languages are capable of implementing the detailed design presented in Appendix E. Although, the C language meets all the requirments for an acceptable language, it is dismissed early because it is not currently supported in the host environment. The remaining two languages FORTRAN and Pascal are compared below.

FORTRAN is a high-level language which was first introduced in the mid-fifties. Since that time it has undergone a steady process of evolution (42). Currently, the host machine, a DEC VAX 11/780, supports VAX-11 FORTRAN version 3.0. VAX-11 FORTRAN is a slightly modified version of FORTRAN-77. In the rest of this discussion VAX-11 FORTRAN will be referred to as simply FORTRAN. FORTRAN primarily uses three control constructs to build its programs, the IF THEN, the DO, and the GOTO (Table VII). The three statements when combined carefully fully satisfy the requirement for modular, logical control constructs. As a

TABLE VII

FORTRAN CONTROL CONSTRUCTS

CHARACTERISTIC	COMMON CONSTRUCT	EXAMPLE USE
LOOPING CONSTRUCTS	DO LABEL NUMBER . LABEL 8 STATEMENT	DO 99 I = 1 TO 20 I = I + 1 J = J - 10 99 CONTINUE
DECISION	IF ... THEN STATEMENT	IF X .LE. 0 THEN GOTO 99
NATURAL LANGUAGE	NOT SUPPORTED	_____
NESTING LEVELS	DO LABEL NUMBER DO LABEL NUMBER STATEMENT STATEMENT LABEL 8 STATEMENT LABEL 8 STATEMENT	DO 89 I = 1 TO 10 DO 99 J = 1 TO 10 K = K - 1 L = L - 1 99 CONTINUE 89 CONTINUE
INTERFACE PARAMETER LIST	SUBROUTINE STUFF(X, Y)	SUBROUTINE STUFF(X, Y)
UNCONDITIONAL BRANCH	GOTO LABEL NUMBER . LABEL 8 STATEMENT	GOTO 99 . 99 IF ... THEN ...

machine independent language (if only the subset of FORTRAN-77 statements are used), its software can be easily rehosted on other machines. If the full capabilities of VAX-11 FORTRAN are used, the language somewhat satisfies the requirement for natural language expressions, by allowing symbolic names to be up to 31 characters in length. However, FORTRAN is rather restrictive with respect to free formatting and it does not support the abstract data type. Finally, FORTRAN does not address the issue of dynamic storage allocation.

Pascal, the other alternative, was designed by

Nickalaus Wirth in 1968 (14). It is a highly structured, type intensive language designed for simplicity and readability. Like FORTRAN, Pascal includes its own versions of the IF THEN, GOTO, and DO statements (Table VIII). Moreover, Pascal combines these constructs with the notion of compound statements. When a group of statements is bracketed by a BEGIN END pair, those statements are logically grouped as one compound statement. Thus, a Pascal program can be viewed as a single statement composed of a number of smaller sub-statements. This structure is very useful when the software engineer is coding for modularity. Because there are few restrictions on Pascal symbolic names, the

TABLE VIII: PASCAL CONTROL CONSTRUCTS

CHARACTERISTICS	COMMON CONSTRUCT		EXAMPLE USE
LOOPING CONSTRUCTS	<pre>WHILE ( ) DO   BEGIN     STATEMENT     STATEMENT   END;</pre>	<pre>FOR COND DO   BEGIN     STATEMENT     STATEMENT   END;</pre>	<pre>WHILE (X &gt; 0) DO   BEGIN     X := X - 1;     Y := Y - 1;   END;</pre> <pre>FOR I := 1 TO 5 DO   BEGIN     X := X - 1;     Y := Y - 1;   END;</pre>
DECISION CONSTRUCT	<pre>IF ... THEN   ELSE</pre>		<pre>IF (X &gt; 0) THEN   Y := 45; ELSE   Y := 34;</pre>
NATURAL LANGUAGE	SUPPORTED FOR VARIABLE AND PROCEDURE NAMES		<pre>VAR SOCIALSECURITYNUMBER PROCEDURE GETPROCESSDATA</pre>
NESTING LEVELS	—		<pre>WHILE (X &gt; 0) DO   WHILE (Y &lt; 0) DO     FOR I := 1 TO 10 DO       BEGIN         J := J + 1;         Y := Y + 1;       END</pre>
INTERFACE PARAMETER LIST	<pre>PROCEDURE NAME (VAR NAME) FUNCTION NAME (VAR NAME)</pre>		<pre>PROCEDURE STUFF (VAR X); OR FUNCTION COMPUTE (VAR Y);</pre>
UNCONDITIONAL BRANCH	<pre>GOTO LABEL NUMBER . . . LABEL 8 IF ... THEN ...</pre>		<pre>GOTO 25 . . . 25 IF ... THEN ...</pre>

requirement for natural language expression is supported somewhat. In addition, Pascal also supports ADT's which aid the software engineer in expressing the program solution in familiar terms. Finally, Pascal can be free formatted and it provides the capability for dynamic storage allocation.

Of the six available languages, only three were actually evaluated as language alternatives. Of those three, Pascal satisfied the requirements for the Test Methodology most satisfactorily; therefore it is the language selected for implementation.

The Implementation Strategy. In the last section of Chapter Three a prioritized list was presented outlining the implemetatation strategy of the Test Methodology. This list was not the implementation strategy, but was written to document incremental and measurable goals which could be satisfied once implementation began. In the implementation phase this list is further detailed, and the strategy which produces this new list is explained by presenting two approaches to program implementation.

The classic approach to implementation is what is referred to as bottom-up programming (66:59). In this type of programming the very lowest modules are first written and tested. Then the next higher level in the program structure is implemented and tested. The process of implementing and debugging the next highest level continues until the

software system is completely implemented and tested.

The second approach to implementation is top-down programming (66:55). In top-down programming the design is implemented in the same top-down fashion in which the design was produced. Once the design is complete (or nearly complete) the code for the main (highest level) module is written. Then the main module's subordinate modules are coded and tested. The process continues until the design is fully implemented and tested.

Both of these implementation approaches have advantages and disadvantages. Bottom-up programming is useful for creating a library of tools or functions, each of which performs a particular function. This idea of a library is expressed best in the quote, "each module should do one thing well" (36:64). However, bottom-up programming is not generally useful as an en masse approach to software design implementation (65:78).

As a follow-on approach to top-down design, top-down programming has a number of advantages as an implementational approach. First, it allows the upper level of the design structure to be implemented early (before the completion of the detailed design). This is important when the user wishes to see how a first version of the software might appear. Second, as modules are added, driver routines are not necessary, because higher levels already exist to

call the new lower level routines. Third, if there is a problem with meeting a deadline for software delivery, then a system which partially fulfills the requirements, or a version that supplies the most needed requirements may be delivered, rather than an un-integrated collection of modules (65:68). Finally, top-down programming supports top-down testing during the implementation phase; this is perhaps the strongest argument for top-down programming for reasons which appear later in this chapter in implementation testing.

The SDW Test Methodology employs both of these approaches in its implementation strategy. The top-down programming approach is applied as the primary implementation strategy for the Test Methodology. This strategy is used in implementing all the upper and middle level modules. The upper level modules are defined to be the the main executive routine and all its immediate subordinates and the modules which implement the menu and help facilities. The middle level modules are the component tests. The name component is given to describe the Test Methodology's tests, because of their highly independent and modularized character.

Stating that a primary strategy exists implies that there must be, by default, a secondary strategy. The secondary strategy for implementation is the bottom-up



approach. Because the best use of bottom-up programming is in providing a low level library of functions, this approach is used to create some of the lowest level functions of the Test Methodology. The benefits of this approach are twofold. First, abstract data types can be implemented at the lowest level and used throughout the Methodology to help make the testing problem addressable in natural terms. Second, a library is created to provide simple graphics and user-oriented screen clean-up routines to aid the software engineer in presenting the software solution in a simpler and more visually appealing manner. With the scope of each of the strategies defined the implementation sequence is discussed.

The main executive module is implemented first in keeping with the top-down programming approach. This module along with its subordinates provides access and initialization of the Test Methodology. Concurrently, as they become needed, terminal cleanup and cursor positioning routines are collected into a small library for use at all levels. This library is supplemented with additional routines throughout the rest of this implementation.

The menu and user interface is implemented next along with the help facility. The menu structure provides the framework for the Test Methodology. It also lets the software engineer see how this early version of the Test

Methodology might appear to a user, and determine what changes are necessary in the interface. The changes which are a result of this initial implementation are presented in the section on Version 1.0 of the SDW Test Methodology. Also, at this stage the abstract data types are implemented so they are available to the component tests which are implemented next.

The last and most major portion of the Test Methodology to be implemented is the test group, the middle level functions. Also, during this stage the I/O interface modules to the INGRES database are implemented. Implementation of these modules is in accordance with the good tools ideas presented in (37). For the component tests any order of implementation is possible due to the independence of each tool group; however the order given below is determined by the implementation schedule and the immediate need for particular tool groups at AFIT:

2) The Consistency Tests - The implementation of the diagnostic tools begins with the requirements and design tests, because these phases are the least understood and automated at AFIT with respect to testing. The consistency tests are selected, because they offer a high return on the amount of time they can save the student in this area of testing.

3) The Correctness Tests - The next step of testing to assure a "good" requirements definition or design.

4) The Clearness Tests - The final class of tests to complete requirements testing. These tests are completed next to provide at least one phase of lifecycle test support.

5) Reconfiguration of the SDW's Resident Diagnostic Tests - After complete reconfiguration of these tests, the final three phases of the lifecycle will be supported. The interest here is to implement a majority of the Methodology in the time remaining.

6) The Tracability and Correctness Tests - Necessary to complete the preliminary design phase of the Test Methodology.

7) The Execution Test/Rehosting of a PDL Processor - The execution test is not currently supported by the necessary software (the PDL Processor). Therefore, it will be added in the Processor support becomes available and time permits.

The Test Plan. In the implementation phase three phases of testing are identified, test plan generation, software execution, and output analysis (31:77). This section is a written report on the SDW Test Methodology Test Plan. Comments on the software's execution are included in the next section, SDW Test Methodology Version 1.0. The Test Plan itself is a high-level discussion of implementation testing methods and techniques, and how a few particular techniques are used to test the Methodology during this phase.

~~Implementation Testing Techniques.~~ Implementation testing techniques can be divided in two ways, either formal and informal, or by correctness and confidence. If the first division is used a spectrum from formal, program verification, to informal, walkthroughs, represents the collection of techniques. If the latter division is used program verification stands alone as the only technique which proves software correctness. The rest of the techniques are grouped together a methods which can only raise the software engineer's confidence that the software is correct. Under either division, there are far too many techniques to be discussed in this investigation; therefore representatives of the spectrum of methods (formal to informal) are presented.

Program verification uses the notion of correctness proofs to prove that a program executes correctly with respect to a complete set of test data (45:45). However, the proofs to prove correctness are often complex even for relatively simple programs. Because of this complexity this method it is not widely used; however it remains a topic of continuing research (11:23, 67).

A more common method in software testing, and a representative of the middle ground of the spectrum, is path testing (34). This technique advocates exercising the program logic so that each path is executed at least once.

A path is a segment of sequential program statements. More formal path testing requires that various combinations of paths be selected during testing. Testing by this and similar methods cannot guarantee that a program is correct, but these techniques can raise the software engineer's confidence toward the software if the method reveals no software errors.

Other techniques also representing the middle ground are functional testing and boundary value analysis. Functional testing demonstrates that the software operates correctly by selecting specific inputs to obtain specific outputs. If the test data is judiciously selected to exercise all facets of the software and the output is what is expected and desired, then the software is functionally correct. Boundary value analysis requires the software engineer to select test cases which represent extremes. For example, If a process can handle up to five inputs, boundary value analysis requires that the process be tested with 0, 1, 5, and 6 inputs. In boundary value analysis errors are assumed to exist near any limiting value. This technique can be useful in determining specific test case data, but generally should be applied in coordination with some other technique.

A representative of the least formal techniques is the walkthrough. The purpose of a walkthrough is to

systematically examine the code for structure, readability, and a cursory examination of program logic. The walkthrough is performed by the software engineer responsible for the code reviewing that code by discussing its operation and structure with his peers. The discussion is a step-by-step progression through the code, hence the name walkthrough.

Selected Methods. During the testing of the Test Methodology four of the above mentioned techniques are employed. To test that the Methodology executes correctly three techniques, path testing, boundry value analysis, and functional testing are used. To test the software's structure and readability a walkthrough is performed of the code.

By applying a combination of methods to test execution, different characteristics of the software can be focused upon. Path testing is used primarily to validate each module for accessibility, and the absence of errors when paths are traversed in different orders. Boundry value analysis is used to determine test cases for validating the Methodology's looping constructs, by causing each loop to be executed zero, one, or n times. This method is also applied in testing the user interface, because users often introduce boundry type errors. Typical errors which can be expected are:

- 1) user commands not in the specified range,
- 2) string commands type too long or too short, and
- 3) no command entered when prompted.

The final method applied to test the execution of the Test Methodology is functional testing. This method focuses on the component tests of the Methodology. Each component test is provided data to simulate normal operation. In addition, erroneous data is provided to validate each tests error handling.

A walkthrough is the last test method applied to the program code. This is done to insure that the code is readable and well structured. Readability includes factors such as indentation, module headers and sufficient in-line documentation. Structure refers to the leveling and breakdown of the solution into modules. It also refers to the manner in which the modules interface with respect to principles of good coupling and cohesion (45).

#### SDW Test Methodology Version 1.0

This section describes the programming standards as they are applied to the Test Methodology software. It also give a cursory discription of the operation of Version 1.0. The section concludes with comments on the status of the Methodology's development.

Programming Standards. "After years of 'write-only code,' students, teachers, and computer professionals now recognize the importance of readable programs" (36:ix). This quote sums up the importance of readability in software. An aid to readability, and itself a good programming practice, is modularity. Readability and modularity are important because they simplify program testing, as well as make easier the maintenance of programs. It is with these results in mind, that the following programming standards are presented:

- 1) Each file is required to have a module header at the beginning of the file with the perscribed format (figure 15). The header information informs the software engineer on what appears in the file and how it is to be used.

- 2) Each module is required to have at the beginning a module header with the perscribed format (figure 16). The header information informs the software engineer or a particular module's dependencies on other modules and data.

- 3) The layout of the program code must show, by indentation, the logical structure of the software. In addition, each BEGIN END block of code must be indented to visually identify it as a compound statement.

- 4) In line documentation is required to be set off by blank lines or written in the right margin (seperated from the code by blank spaces).



```

(*****
* DATE: The date in which this file is created *
* VERSION: The current file version number (e.g. 1.0) *
* TITLE: A descriptive name of what this file contains *
* FILENAME: The name under which this file is accessed *
* OWNER: The creator of this file *
* SOFTWARE SYSTEM: The software project of which this file *
* is a part *
* OPERATING SYSTEM: The operating system this file is *
* created under *
* LANGUAGE: The type of any code found in this file *
* USE: How is this file used with any others (e.g. linked *
* with) *
* CONTENTS: What modules does this file contain *
* FUNCTION: A brief description of the purpose of this *
* file *
*****)

```

Figure 11: Standard File Header

```

(*****
* DATE: The date in which this file is created *
* VERSION: The current file version number (e.g. 1.0) *
* NAME: The name by which this module is called *
* MODULE NUMBER: The number of the module *
* FUNCTION: A brief description of what this module does *
* INPUTS: A list of the input parameters *
* OUTPUTS: A list of the output parameters *
* GLOBAL VARIABLES USED: Explanatory *
* GLOBAL VARIABLES CHANGED: Explanatory *
* GLOBAL TABLES USED: Explanatory *
* GLOBAL TABLES CHANGED: Explanatory *
* FILES READ: A list of all the files accessed by this *
* module *
* FILES WRITTEN: A list of the files written by this *
* module *
* MODULES CALLED: All modules called by this module *
* CALLING MODULES: All modules which call this module *
* AUTHOR: The creator of this module *
* HISTORY: A record of versions, their authors, and the *
* dates they were created *
*****)

```

Figure 12: Standard Module Header

5) Separate files are required for each major division in the Test Methodology's structure. This is done for efficiency in compilation and to promote modularity. The division as they exist are as follows:

- a. the executive structure,
- b. the menu structure,
- c. the component tests,
- d. the data access routines,
- e. the supporting libraries, and
- f. the data files.

6) Descriptive variable names are required.

The application of these standards promotes modularity and readability in the Test Methodology. As an introductory and evolving system the Test Methodology needs these qualities to simplify the enhancements and maintenance which may occur.

Version 1.0. The SDW Test Methodology Version 1.0 is a prototype of what the Methodology was originally designed to be. Version 1.0 includes has implemented the executive structure, the menu structure, the first of the component tools, a subset of the data access routines, the supporting libraries, and the menu definition and help files. Each of these divisions of the methodology are discussed in greater detail in the following sub-sections.

Executive Structure. Version 1.0 is controlled at the top-most level by the executive structure. The modules which make up this structure are stored in the file MAIN.PAS. These modules are responsible for providing the

introductory template, prompting the user for the project to be tested, and placing the user under the menu structure's control.

Menu Structure. Two major functions occur during the period in which the user executes the menu structure routines, the menu is generated and the interactive process begins. Construction of the menu depends entirely upon the definition provided for in MENU.DAT. Any changes to this file require similar changes to be made in procedure WRT\_OPTION if the new or modified menu item is to be seen on the terminal screen. If WRT\_OPTION is not modified the new selection may be accessed; although, it does not appear as an option. More information on menu modifications for the addition/deletion of component tests is provided for in the section on enhancements to the Test Methodology. Once the menu is completely defined, the user is placed at the top-level and the interactive process begins.

Interaction with the Test Methodology is simple and straightforward. A menu of options is first displayed. Then the user is prompted for a response. If the response is one of the menu options, then this dialog continues until the desired component test is selected (the bottom of the menu) or the program is exited (the top of the menu). If an invalid response is given the user is sent a warning message and re-prompted. At any point in this dialog the user may

request help on a particular option. After a test is selected and executed, the user is placed in the menu at the point at which the test was entered.

Component Tests. Version 1.0's menu is implemented to support 24 tests and seven test related functions; however only the first component test, ver\_nam.num, is presently implemented. As the only existent test it sets the standard for each of the tests to follow. Principle features which it is expected to be used as a standard for are interfacing, data access, documentation, program control, and function content.

Data Access Routines. Version 1.0 is initially developed to interface with the INGRES Relational Data Base System. However, because the Test Methodology is designed to be independent of a particular database, the data access routines to INGRES are modularized and organized into their own separate file. In this manner, different sets of data access routines can be supplied to Version 1.0 for different databases.

Support Libraries. Version 1.0 is provided with two libraries for its operation. The first and most important in the library of data structure manipulation routines is the file DTASRT.PAS. These routines provide Version 1.0 with the capability to store the data returned to its tests by INGRES. The second library is in the file

VTLIB.PAS; this library contains screen clean-up, cursor positioning, and simple graphics routines to make Version 1.0 more user friendly and attractive.

Data Files. Two data files are included with Version 1.0, they are MENU.DAT and HLPFILE.DAT. MENU.DAT contains the menu description as previously mention. HLPFILE.DAT contains all the text information for the Version 1.0's help facility.

There are four points which are noteworthy concerning the structure of Version 1.0, and they all stem from the decision to divide the Methodology into modular units. They are:

- 1) the simplicity in which component tests can be added,
- 2) the independence of the methodology from a particular DBMS,
- 3) the ease in which the underlying data structure may be changed, and
- 4) the flexibility provided in a text file menu description to support the simplicity of the addition/deletion of component tests.

Figure 17 is provided as a graphic representation of the division of the Test Methodology into separate files. Each solid line rectangle represents files which are already implemented as part of Version 1.0; each dashed line rectangle represents a file of modules which would be needed to complete the SDW Test Methodology according to its design.

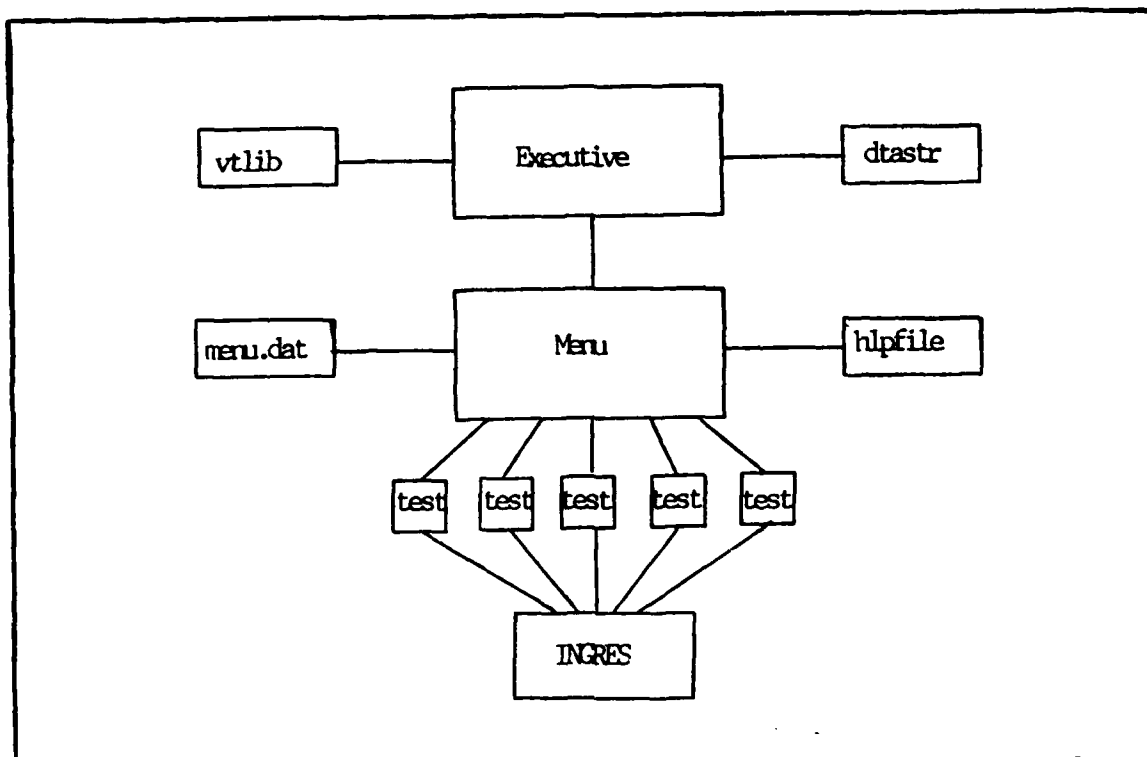


Figure 17: SDW Test Methodology as a Modular System

### Summary

The implementation phase is the forth phase in the software lifecycle. It is during this phase that the software is produced. However before implementation can begin the software engineer needs to address issues concerning the language selection, the implementation strategy, and the test plan. Once these issues are resolved the software engineer may begin the coding process.

This chapter is the record of the implementation phase for the SDW Test Methodology. It begins by addressing each of the three issues mentioned above; it concludes with a

discussion on Version 1.0 of the Test Methodology. Four noteworthy characteristics are identified in Version 1.0; they are all a result of the division of the Methodology into modular unit files.

## Recommendations and Conclusions

### Introduction

The purpose of this investigation is to examine lifecycle testing and to produce a test methodology to support the software engineer throughout the entire lifecycle. To reduce the solution domain, an emphasis is placed on testing in a software development environment and to testing by automated tools. Because the Test Methodology produced in this investigation is a software product, the format of this investigation follows the lifecycle development model. This model is characterized by its development phases, each of which corresponds to a chapter in this document.

### Recommendations for Future Development

The Test Methodology as conceived can be a useful aid in software testing. Although considerable design structures have been developed, the component tests are not supported at present. The original goal was to support at least the requirements phase; however, time only permitted the interfaces for this support to be put in place. What is currently implemented is the menu structure and its library of support files. This software is the hub of the Methodology (see figure 17). With this hub in place, groundwork is laid for follow-on work to continue.



The Component Tests. The component tests determine the character and capabilities of the Methodology. This document only suggests the types of tests which should be performed; a follow-on investigation could add those tests which would be particularly useful in the software engineer's environment. For the AFIT environment it is recommended that tests supporting the requirements and design phases first be added, since currently the testing at this level is mostly manual.

The Data Access Routines. The Methodology is originally designed to interface with the INGRES data base system; however, because it is to be independent of INGRES, the routines to interface with INGRES are packaged as a set of data access routines. These routines have not been implemented yet. The implementation of these routines and other packages to make the Methodology useful with different DBMS's is also recommended.

Redesign of the Methodology. The Methodology is designed to be an aid to testing in software development environments, in particular the AFIT environment. In the production of the Methodology's design there were decisions which were made which had more than one option. An effort was made in these situations to address the alternatives and to justify the decision made. Undoubtedly, many of the alternatives were valid directions to pursue. As a first

attempt to support AFIT software testing, this Methodology can be improved; a objective and brief redesign of the Methodology is recommended as a starting point for follow-on investigations to encourage the production of an improved Test Methodology.

### Conclusions

As long as software remains an integral and critical part of technology and human lifestyle the need for it to be accurate remains. The development of the Test Methodology is an example of an evolving solution to a particular problem. As an evolving solution, both the original purpose of the Test Methodology and its scope have changed. These changes were brought about as the author's understanding of software testing grew during the course of this investigation. The conclusions which are presented below reflect this evolving understanding.

It became evident early in the investigation that software testing could not be completely automated. This is particularly true in the requirements and design phases, because of the need for ambiguity in top down analysis and design. Therefore, automated testing tools should not strive to be a panacea for testing, but they should rather be designed as an expert aid to the software engineer in testing. As the details for the software are identified, a greater portion of the burden of testing software accuracy

can be placed on the automated aids; however, in the early phases of the lifecycle, validating the correctness software will continue to be an issue which must be determined by people. The Test Methodology has been designed according to this objective; that it be a useful aid in testing to software engineer with a varying range of experience levels. Emphasis is placed on the Test Methodology as an aid to testing and not the answer to testing in software development environments.

It is also noted that the Test Methodology, indeed any test aid, must be adaptable to its environment. The Test Methodology was created for the AFIT environment. As soon as a particular environment is targeted for the development there is a tendency to design to that environment. Hopefully, this siduation has been avoided by making the Methodology as modular and independent of other software as possible.

#### Summary

The Test Methodology is an initial attempt to evaluate the requirements for supporting lifecycle testing. As an aid, it only provides the outline for the software which must come later. The ideas presented herein are useful in understanding what can and cannot be tested. More importantly, they point out for a particular environment, the AFIT environment, what types of support are currently

needed to support AFIT software testing. For the author, it was a lesson in research and analytical thought. Furthermore, it strenghten the author's understanding of how broad and complex, how widely written and yet still fuzzy the topic of software testing is.

APPENDIX A

Requirements Definition Model

## SDW Test Methodology

### Node Index

- A-0 Perform Lifecycle Test Methodology
  - A0 Perform Lifecycle Test Methodology
    - A1 Perform Requirements Tests
      - A11 Perform "Correctness" Tests
        - A111 Examine Leveling
        - A112 Verify Active Process Names
        - A113 Record Data Input/Output Flow
        - A114 Test for Missing Text
      - A12 Perform Clearness Tests
        - A121 Count Processes
        - A122 Count Data
        - A123 Check for Notational Abberviations
      - A13 Perform Consistency Tests
        - A131 Verify Diagram Name and Number
        - A132 Verify Interfaces
        - A133 Verify Internal Consistency
        - A134 Record Aliases
    - A2 Perform Detailed Design Tests
      - A21 Perform Design "Correctness" Tests
        - A211 Examine Leveling
        - A212 Verify Active Process Names
        - A213 Record Data Input Output Flow
        - A214 Test for Missing Text
      - A22 Perform Design Clearness Tests
        - A221 Count Processes
        - A222 Count Data
        - A223 Check for Notational Abbreviations
      - A23 Perform Design Consistency Tests
        - A231 Verify Diagram Name and Number
        - A232 Verify Interfaces
        - A233 Record Aliases
      - A24 Perform Traceability Tests
        - A241 Match Dictionary Entries
        - A242 Verify Unmatched Entries
        - A243 Build Traceability Matrix
      - A25 Perform Design Completeness Tests
        - A251 Retrieve Requirements Model
        - A252 Retrieve Design Model
        - A253 Compare Functional Intent

- A3 Perform Detailed Design Tests
  - A31 Perform Design Correctness Tests
  - A32 Perform Design Clearness Tests
  - A33 Perform Design Consistency Tests
  - A34 Perform Traceability Tests
  - A35 Perform Design Completeness Tests
  - A36 Begin Execution Tests
    - A361 Read Algorithms
    - A362 Concatenate Algorithms
    - A363 Provide Editing
    - A364 Execute Det Des Program
- A4 Perform Code Tests
  - A41 Provide Static Tests
  - A42 Provide Dynamic Tests
  - A43 Provide Editor
  - A44 Perform Traceability Tests
- A5 Perform Integration Tests
  - A51 Provide Static Tests
  - A52 Provide Dynamic Tests
  - A53 Provide Editor
  - A54 Perform Traceability Tests
- A6 Perform Operations and Maintenance Tests
  - A61 Provide Static Tests
  - A62 Provide Dynamic Tests
  - A63 Provide Editor
  - A64 Perform Traceability Tests

## Introduction

The purpose of the Requirements Model is to functionally represent the requirements outlined in Chapter Two. The Structured Analysis and Design Technique (SADT) is selected to represent the SDW Test Methodology's Requirements Model. This Model is presented herein together with a text description of each SADT diagram. This Model and Chapter Two, together, define the requirements for the SDW Test Methodology.



USED AT:	AUTHORITY: Keith A Shopper PROJECT: SDUTM	DATE: 10/31/84 REV: 1	WORKING DRAFT	READER	DATE	CONTENT:
NOTES: 1 2 3 4 5 6 7 8 9 10			RECOMMENDED X PUBLICATION			

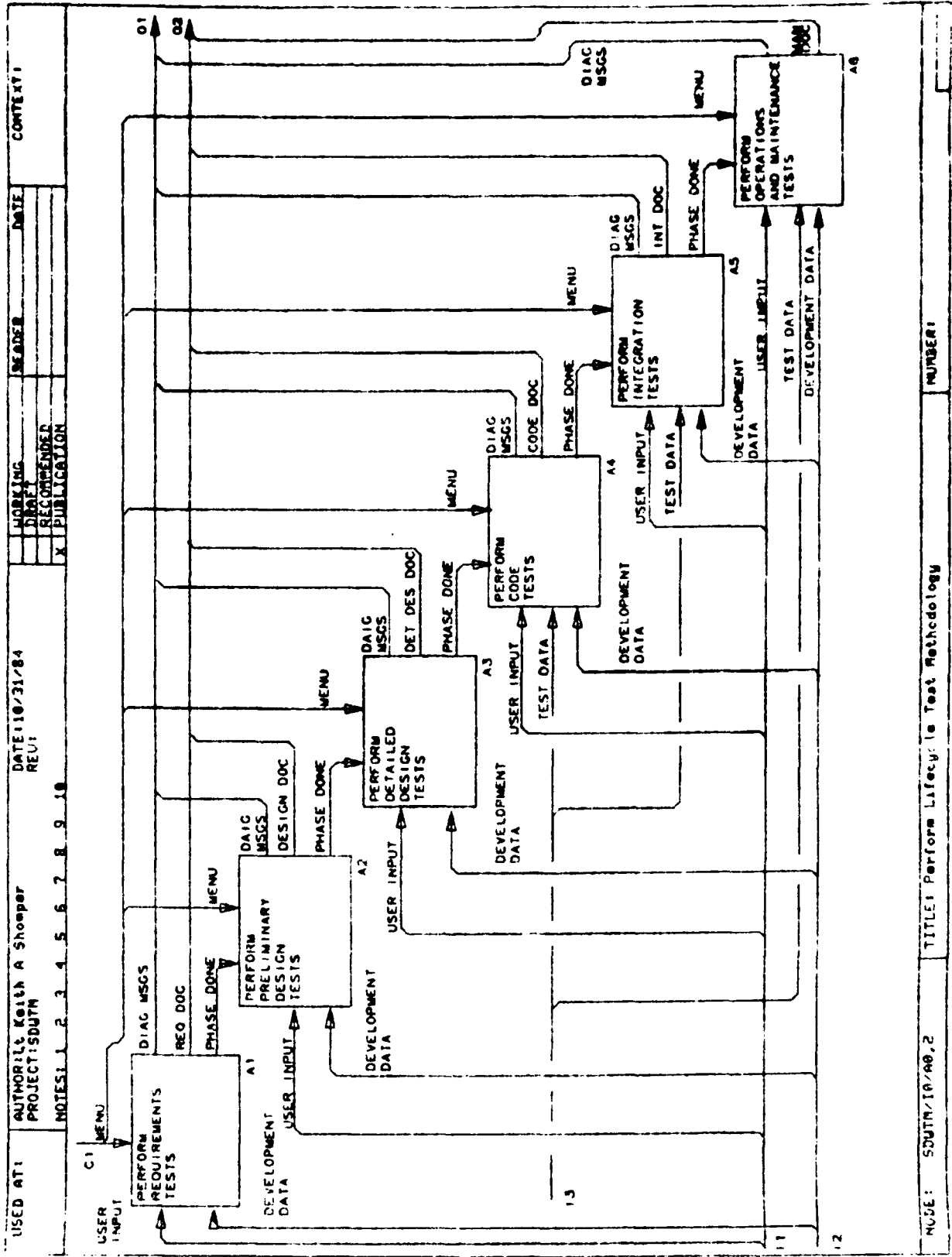
```

graph LR
    TM[TEST METHODOLOGY MENU] --> P[PERFORM LIFECYCLE TEST METHODOLOGY]
    UI[USER INPUT] --> P
    DD[DEVELOPMENT DATA] --> P
    TD[TEST DATA] --> P
    P --> DM[DIAGNOSTIC MESSAGES]
    P --> DOC[DOCUMENTATION]
    P --> DM2[DEVELOPMENT MACHINE]
  
```

MODE: SCUTM/10/80.1	TITLE: Perform Lifecycle Test Methodology	NUMBER:
---------------------	-------------------------------------------	---------

#### A-0 Perform Lifecycle Test Methodology

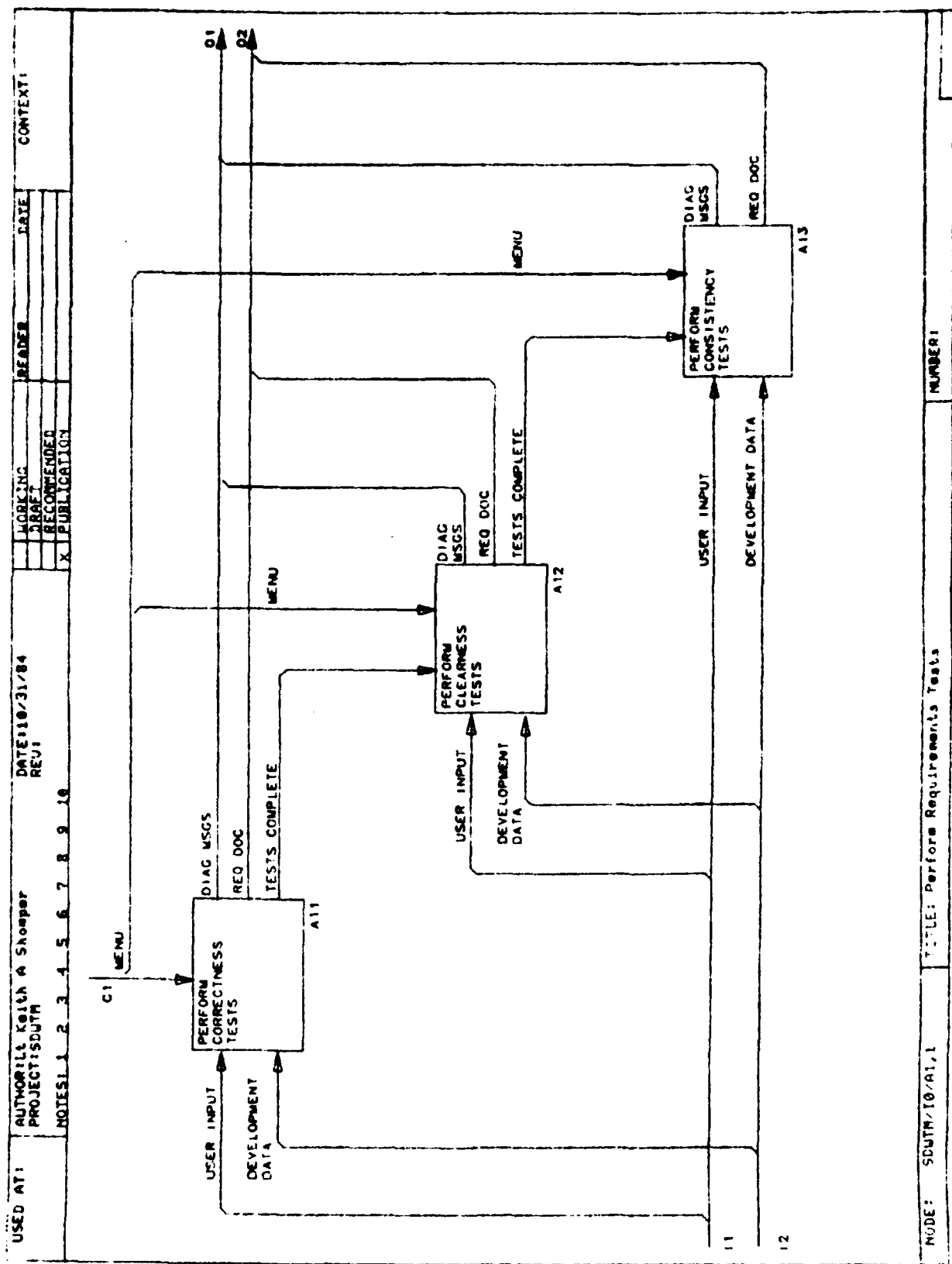
Perform Lifecycle Test Methodology is the top-most level of the SDW test methodology functional model. This level represents the high-level data flow (mass inputs and final outputs) through the system. It also defines system control and hardware support. In this diagram, as in all the others, the data associated with each of the arrows are not explained or defined in the accompanying text, but it are defined with the processes in Appendix D, the Data Dictionary.



#### A0 Perform Lifecycle Test Methodology

That this diagram appears very similar to the software lifecycle presented in Chapter One (figure 2) is no accident. As the software is developed over the course of the lifecycle, testing should occur (55). This diagram reflects this concern for the integration of development and testing. Each process is required to generate documentation that aids in development and serves as a complete history of that development. The diagnostic messages and user input are the interactive modes of communication; they are necessary to inform the user of the development status as it progresses. On-line development allows for rapid evaluation and modification of the development and is oftentimes necessary during the executional stages of program development.

To add flexibility to the test methodology the test procedures may be executed beginning at any of the six process boxes. It is noted here that the proper development data must be resident in the project database for the tests to execute successfully (e.g. requirements data must have been developed if the user intends to begin at the requirements testing level). The process selection feature is available through the SDW menu facility.



NODE: SDUTH/10/A1.1 TITLE: Perform Requirements Tests

NUMBER:

## A1 Perform Requirements Tests

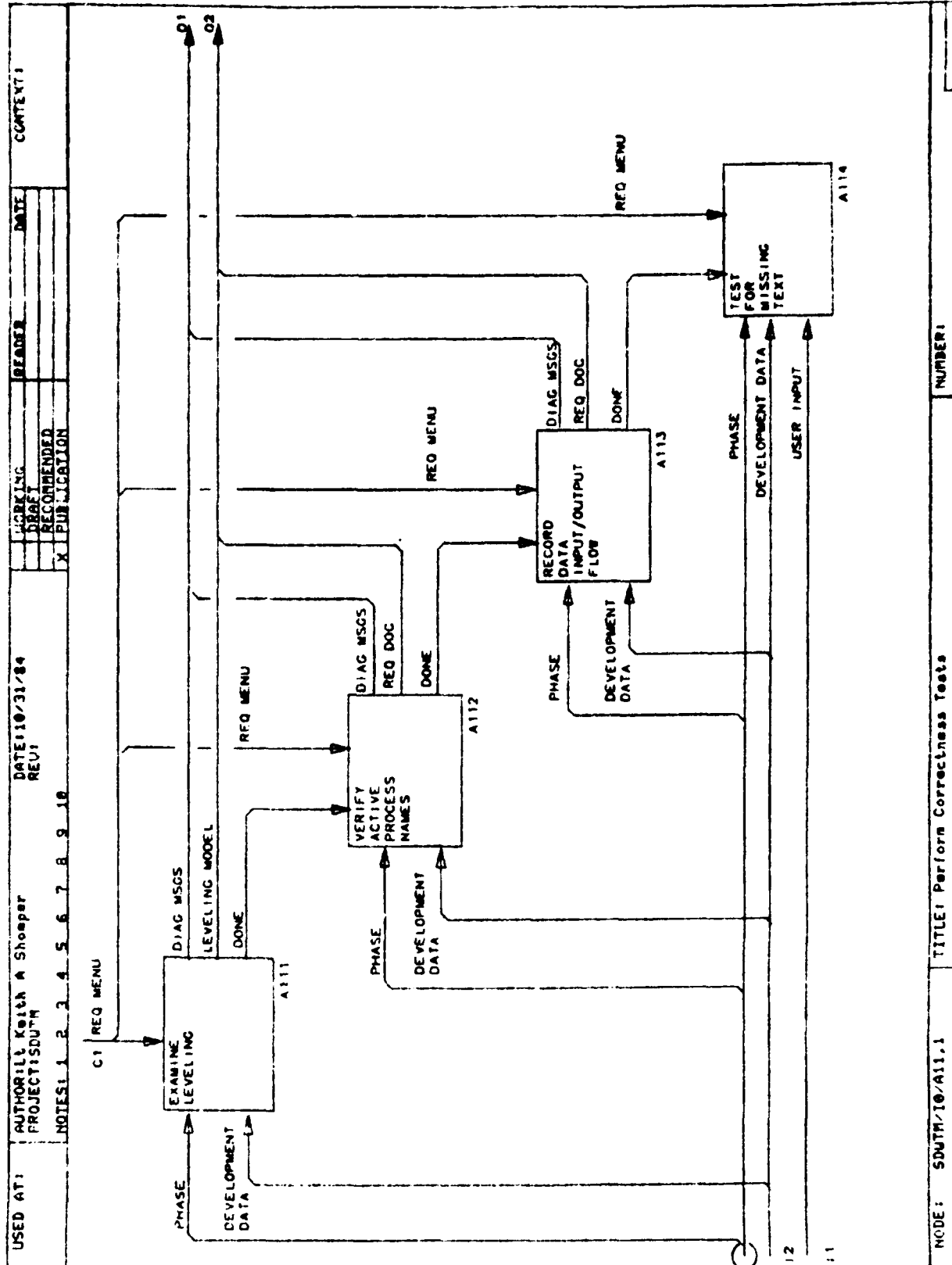
In an effort to determine the essential qualities for software requirements many papers have been written (31, 39, 54, 60). The four qualities cited that are common to all these papers are correctness, completeness, consistency, and clarity. This diagram addresses two of these qualities, as defined by the authors; they are consistency and clarity. Consistency refers to the relationships between diagrams such as interfaces or diagram names. Clarity is more subjective; it is tested by defining a standard measurement of clearness, and then testing by some scale against that standard. A typical test of clearness is how many processes are present in a single diagram (51).

Correctness and Completeness are somewhat more difficult to automate because they rely on semantic interpretation and hierarchy. Correctness is a measure of how well the requirements meet the "real" needs of the user. Since the requirements define the meaning and the purpose of the proposed software, it becomes a standard by which correctness can be measured. It cannot be logically tested against user needs at the requirements level, because it defines the user's needs.

Completeness is a measure of whether or not all the information needed for the proposed software is present in the requirements. This property cannot be tested at the

requirements level because there is no "upper level" specification (the hierarchy problem) from which "all" the information was derived. Trying to test completeness at this level is like building a jigsaw puzzle; if a piece of the puzzle is missing it will not be apparent until the missing piece is needed.

Completeness is not tested for at this level. In subsequent levels where traceability to upper levels is possible functions are available to test completeness. Correctness in this investigation refers to syntactic correctness. This definition is different from the definitions offered by the referenced authors and its general usage; therefore when referred to by this definition it is set off by quotes. Typical of "correctness" testing is an examination of the notational syntax.

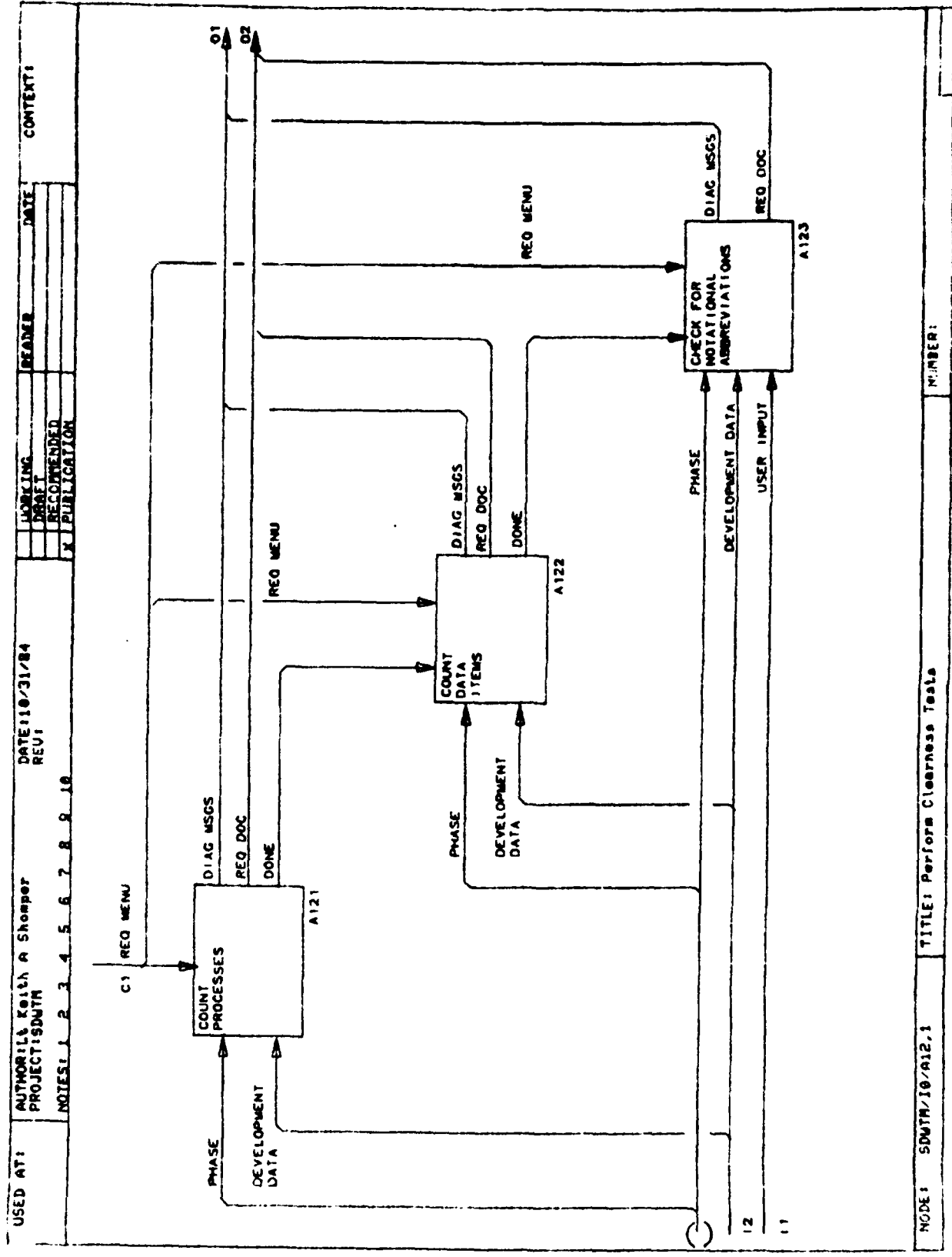




### All Perform "Correctness" Tests

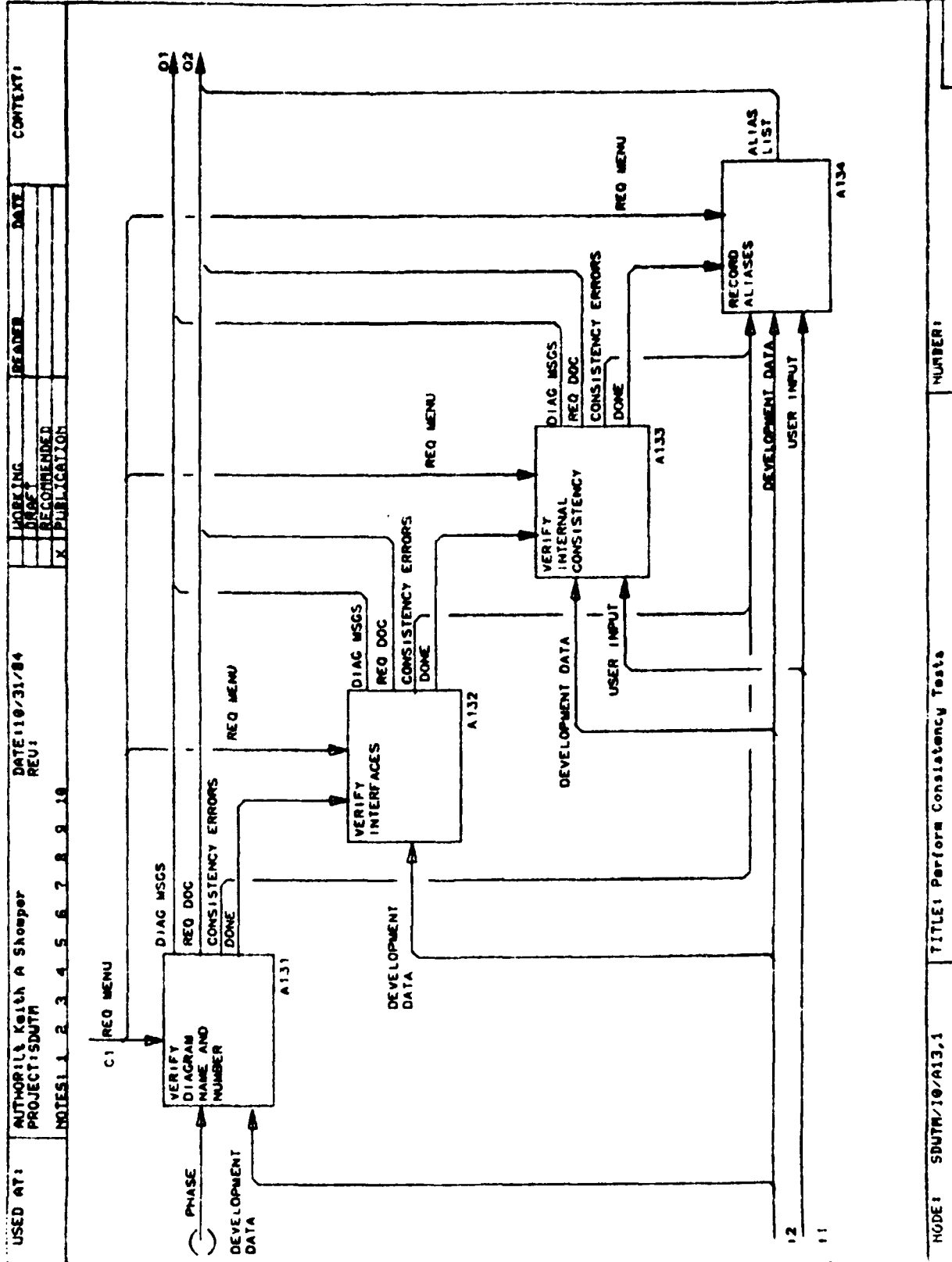
"Correctness" in this investigation refers to syntactic rightness. One measure of "correctness" testing is leveling. Leveling is best represented by a tree structure, and serves as a measure of proper functional decomposition. The idea behind leveling is that a well built software structure will be well balanced. "Well balanced" is defined here to mean, that the depth of a branch in the software structure may not exceed two more than a branch with the same parent node (it is assumed here that the reader has some familiarity with tree structures). If the well balanced rule is violated, then the user is warned with a diagnostic message that a leveling problem might exist.

Another measure of "correctness" is adherence to the use of imperative verbs. By enforcing the use of the imperative verb in the process names, emphasis is placed on what is being done rather than how it is being done. Other measures of requirements correctness are adherence to notational rules.



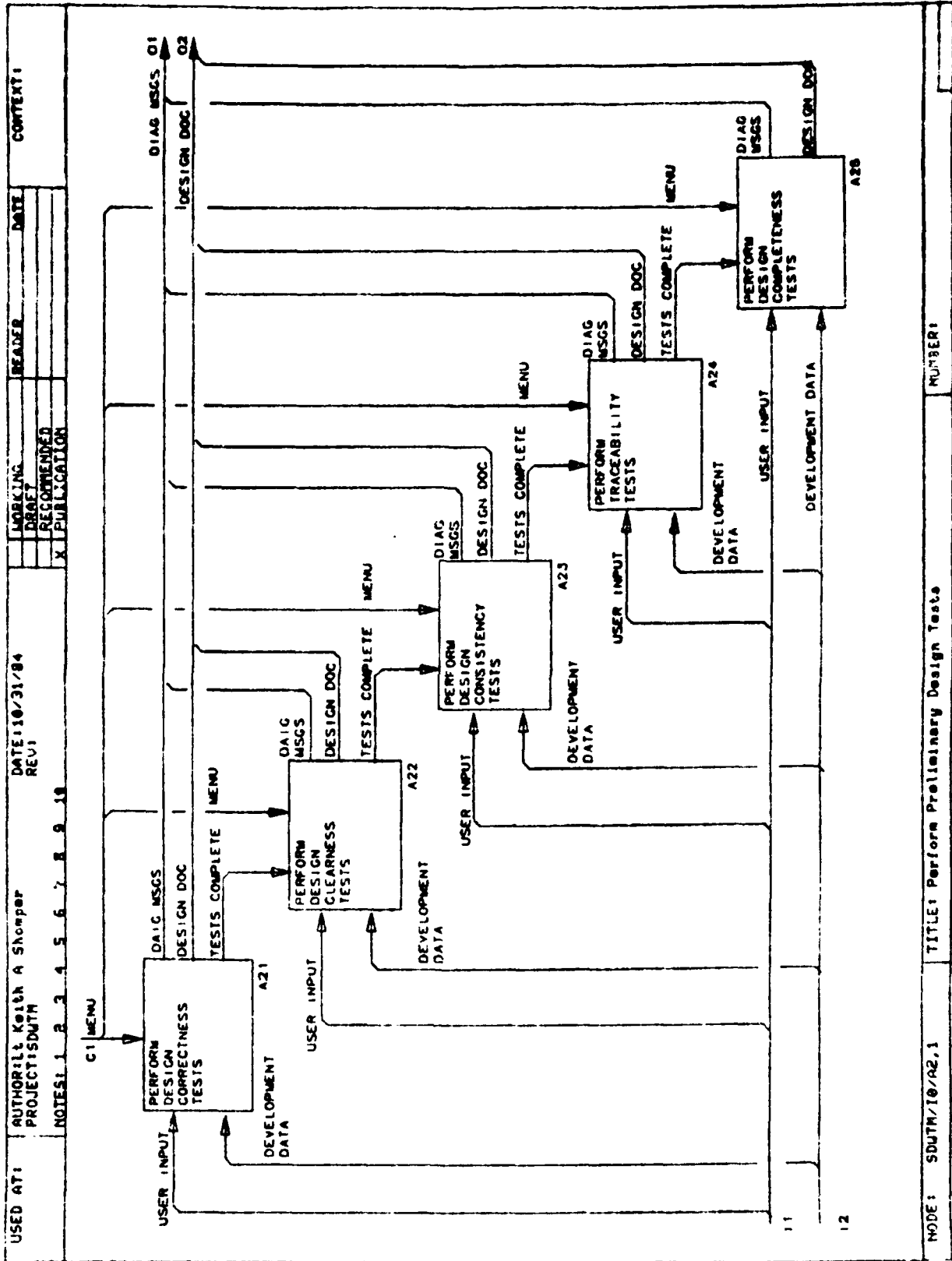
## A12 Perform Clearness Tests

The second of the three concerns for requirements testing is clearness testing. There are no strictly defined rules outlining what is and what is not clear, but there are "rules of thumb" which are sometimes applied to software engineering to limit complexity and enhance clarity. The first is to limit the number of functions into which a process can be partitioned; this number is six or fewer (51). If a diagram contains greater than six processes a warning message is sent to the user indicating that clarity might be compromised by the diagrams partitioning. This six or less rule is also applied to the data flow (input and output) associated with each process node. A second consideration concerning clearness is the dotted-arrow convention used in SADTs. This is an abbreviated notation for feedback arrows, and like most abbreviation it sacrifices clarity for brevity. Thus, when abbreviation such as the dotted-arrow are used, the user will be notified through diagnostic messages that a potential point of confusion may exist.



### A13 Perform Consistency Tests

The definition for consistency in this investigation refers to compatibility or agreement. The purpose of consistency tests is to validate agreement between the "borders" of separate diagrams or modules. Examples of borders are the data flow arrows which cross separate diagrams or the interfaces on associated modules. Consistency tests may also be executed on intradiagram or intramodule information, but experience shows that consistency suffers most from physical separation during development (e.g. separate pages for related diagrams/modules). Perform Consistency Tests looks for these "across-the-border" errors. It also keeps a record of alias names so that what would be flagged as a error is not reported if the "error" is a valid alias.



## A2 Perform Preliminary Design Tests

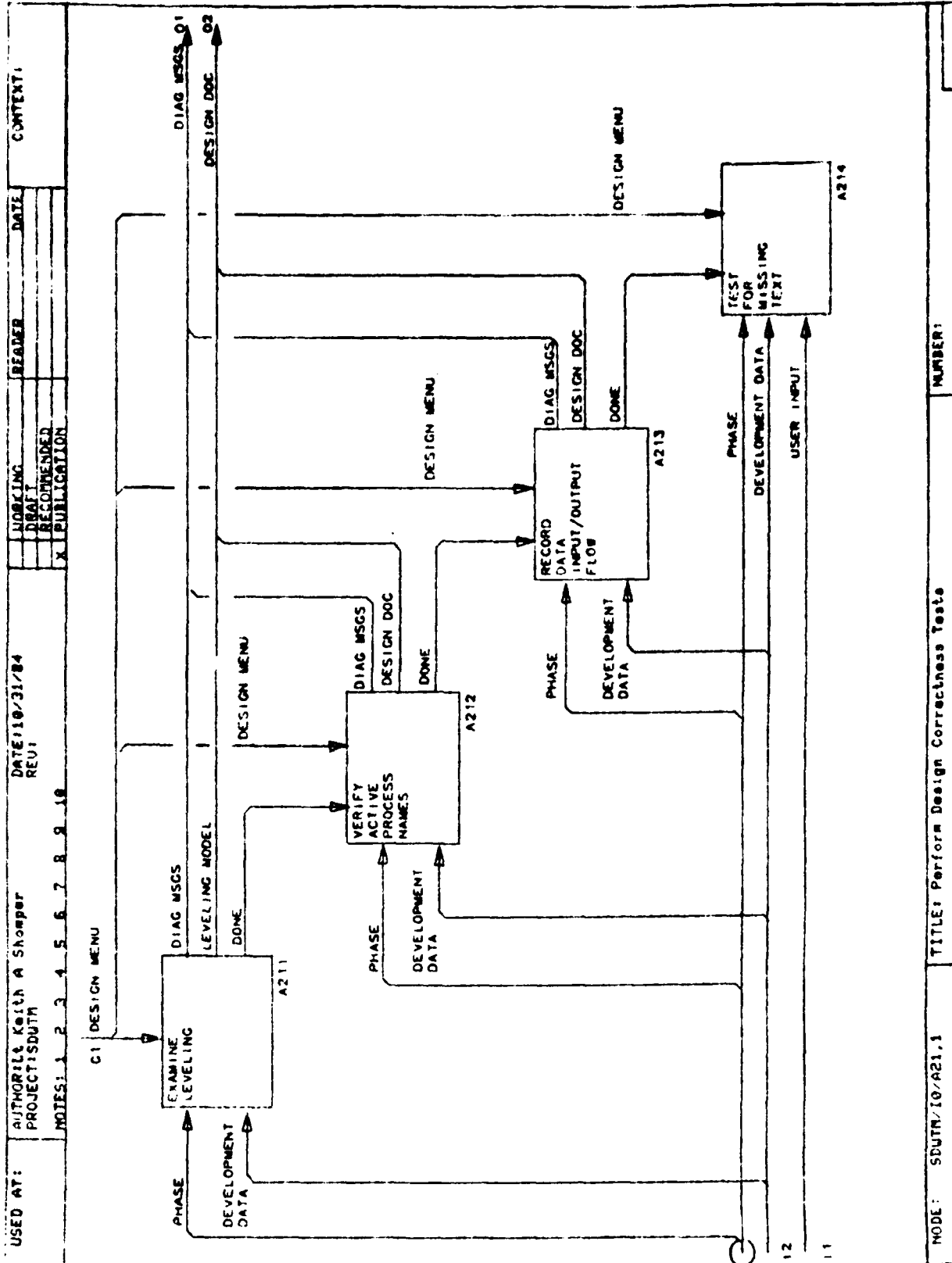
During the preliminary design phase much of the same type of errors will occur as in the previous phase, because of the similarities of the requirements and design graphical techniques. This necessitates similar testing for "correctness", clearness, and consistency during the preliminary design phase. Because the preliminary design succeeds the requirements definition (assuming that the software lifecycle is being used for development), there is now an "upper-level" which the design can be tested against for traceability and completeness.

Traceability checks for information consistency across the phase borders, that is consistency between the requirements and design models. During this consistency check a traceability matrix is generated to link together data from different phases so that it may be traced from requirements to the code level or vice versa. The traceability matrix is first generated at this stage of development and is edited as the development proceeds through each of the following phases.

Completeness is a test for functional similarity between the phases. Testing for completeness is necessary to insure that all the functional capability defined in the requirements phase is supported by the design and then by the code. Since at this level function may not be tested by

execution, the "real" functional capability is presented by the module definition (what they say they can do). Therefore, testing at this point is interactive with the user to determine if the functional capability between phases is being maintained by providing on-line queries of the development database and documentation support.





MODE: SDUTM/10/A21.1

TITLE: Perform Design Correctness Tests

NUMBER:

#### A21 Perform Design "Correctness" Tests

Many of the tests associated with the "correctness" tests of the requirements definition phase are also applicable to the design phase; however, there are differences. At this level of development the flow of data begins to become important, especially with regard to its generation and consumption. Tests on data during the preliminary design should insure that data passed to a module is either passed through that module or consumed by it, and data passed from a module is either passed through that module, sent from another, or generated by that module (31:74).

USED AT:	AUTHOR: Keith A Shooper	DATE: 10/31/84	REVISION:	WORKING	DATE	CONTEXT:
PROJECT: SDUTM			REV:	DRAFT		
NOTES: 1 2 3 4 5 6 7 8 9 10				RECOMMENDED		
				X PUBLICATION		

```

graph TD
    Start(( )) -- 11 --> Split(( ))
    Split --> A221_1[A221  
COUNT PROCESSES]
    Split --> A221_2[A221  
COUNT DATA ITEMS]
    
    A221_1 --> Out1[DIAG MSGS]
    A221_1 --> Out2[DESIGN DOC]
    A221_1 --> Out3[DONE]
    Out3 -- C1 --> Start
    
    A221_2 --> Out4[DIAG MSGS]
    A221_2 --> Out5[DESIGN DOC]
    A221_2 --> Out6[DONE]
    
    Out1 --> A223[A223  
CHECK FOR NOTATIONAL ABBREVIATIONS]
    Out2 --> A223
    
    A223 --> Out7[DIAG MSGS]
    A223 --> Out8[DESIGN DOC]
    Out7 --> Start
    Out8 --> Start
    
    A223 --> Out9[USER INPUT]
    Out9 --> Start
    
    Start -- 12 --> A223
  
```

NODE: SDUTM/IO/A22,1	TITLE: Perform Design Clearness Tests	NUMBER:
----------------------	---------------------------------------	---------

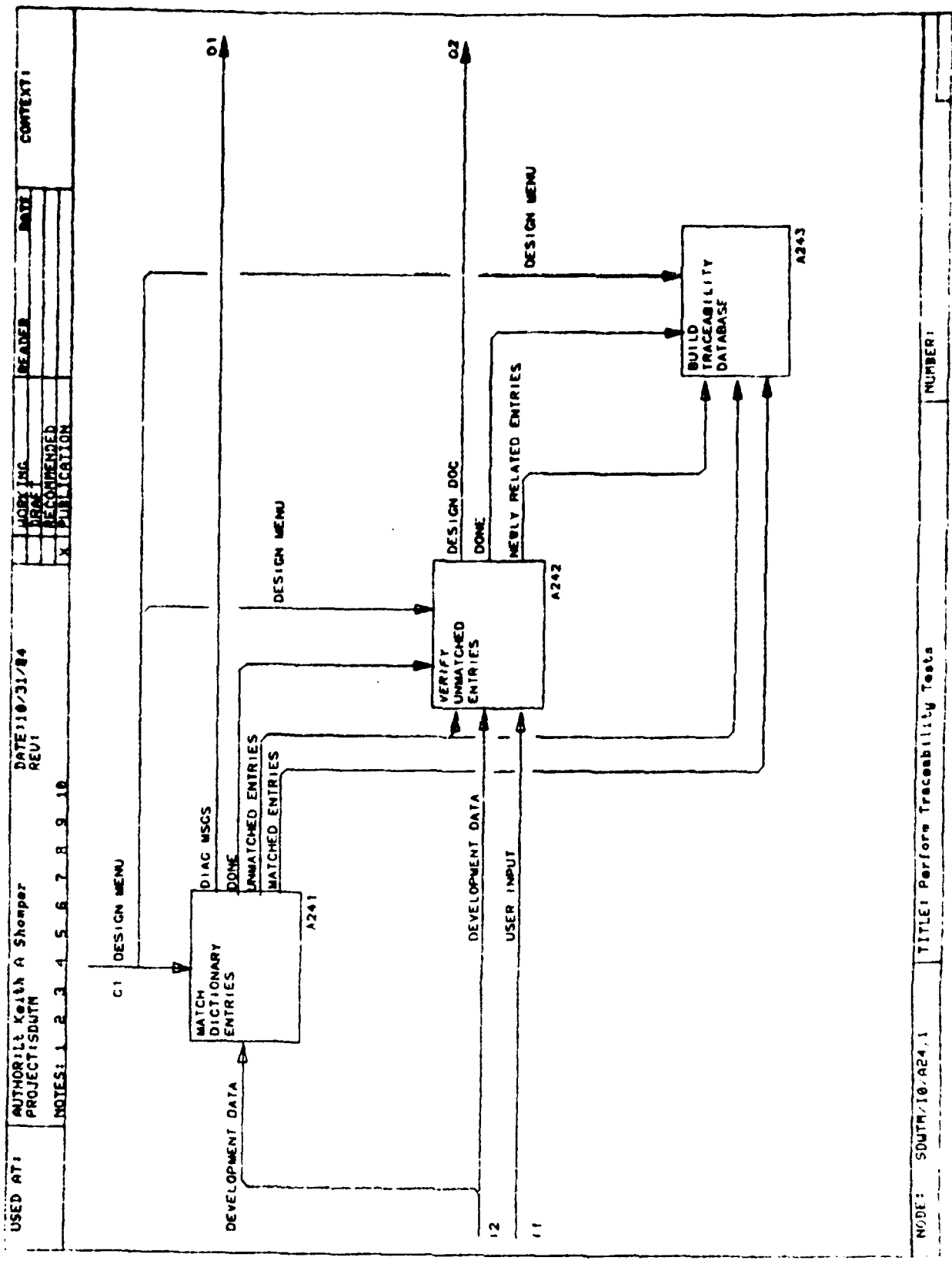
## A22 Preform Design Clearness Tests

Excessive process or data flow can quickly cause confusion in interpreting a diagram; therefore, a ceiling of six entities is set for either processes or input/output data. If this ceiling is exceeded the software engineer is warned that obtuseness might exist in the design diagram. Another opportunity for misinterpretation occurs whenever shorthand or non-standard notation is used (e.g iteration arrows, decision diamonds). Because some of the shorthand notation is very useful, the test permits its use, but it reminds the software engineer that a possible point of ambiguity exists.



### A23 Perform Design Consistency Tests

Structure charts are a graphical design technique. They are similar to SADTs and DFDs in that they all show functional capability, hierarchy, decomposition, and data flow, and they are all graphical. Because they are in many respects similar, they can be tested similarly for consistency; therefore, despite the different representational schema, the functions' purposes remain the same.



#### A24 Perform Traceability Tests

Traceability from the requirements to the code is important and necessary in determining if the requirements have been satisfied. Traceability begins in the preliminary design, because it is the first phase with an "upper-level" to test against.

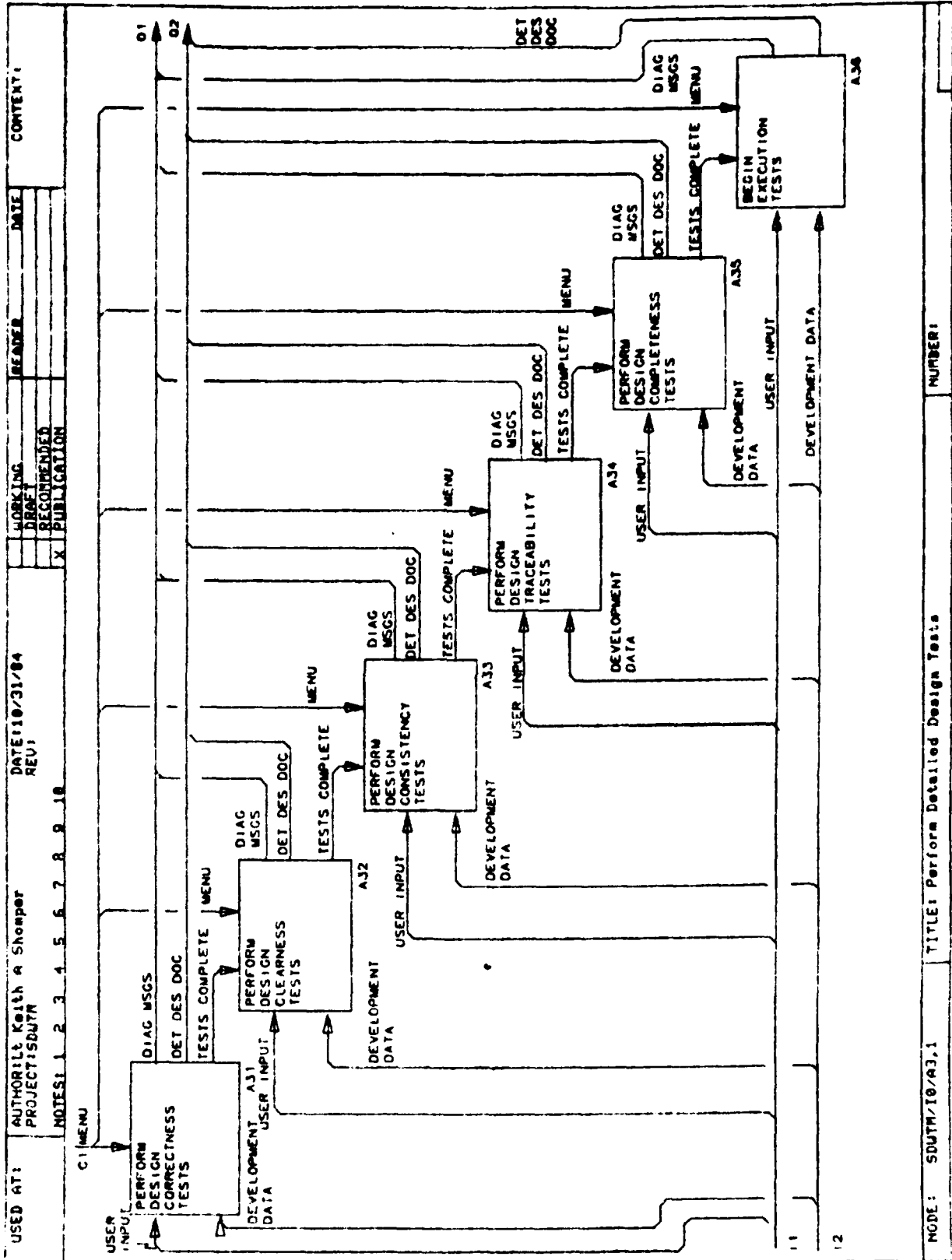
In this process, Perform Traceability Tests, each process and data item from the preceeding phase of development is related to a process or data item in the current development phase. The relation is documented in the traceability martix. When an association is not found for a process or data item the software permits the user to interact with the development database to try and establish a logical association. If no association or relation for that process or data item is determined then its originality is documented and the item is added as a new item to the traceability matrix. The traceability matrix is complete after all phases of development (up to coding) are complete.





## A25 Perform Design Completeness Tests

The purpose of the completeness tests is to insure a total transfer of functional capability from one graphic technique (SADTs or DFDs) to another (structure charts). Much of this testing is done by the traceability tests, but the final test is an intelligent examination by the user of the function definitions in each phase. This examination cannot be automated, because meaning must be extracted from the function definitions in order to judge whether the functional capability is preserved. Perform Design Completeness Tests provides the user access to the development database for these tests and documents his conclusions concerning completeness at the time they are made.



MODE: SDUTN/10/A3.1

TITLE: Perform Detailed Design Tests

NUMBER:

### A3 Perform Detailed Design Tests

One of the reasons for choosing structure charts was that they could be used in both the preliminary design and the detailed design phases. The SDW test methodology supports this consistency in design, because it reduces the complexity of testing in the design phases by requiring testing for only one type of representation. Therefore, the tests used during the preliminary design are identical to those needed in the detailed design with one exception.

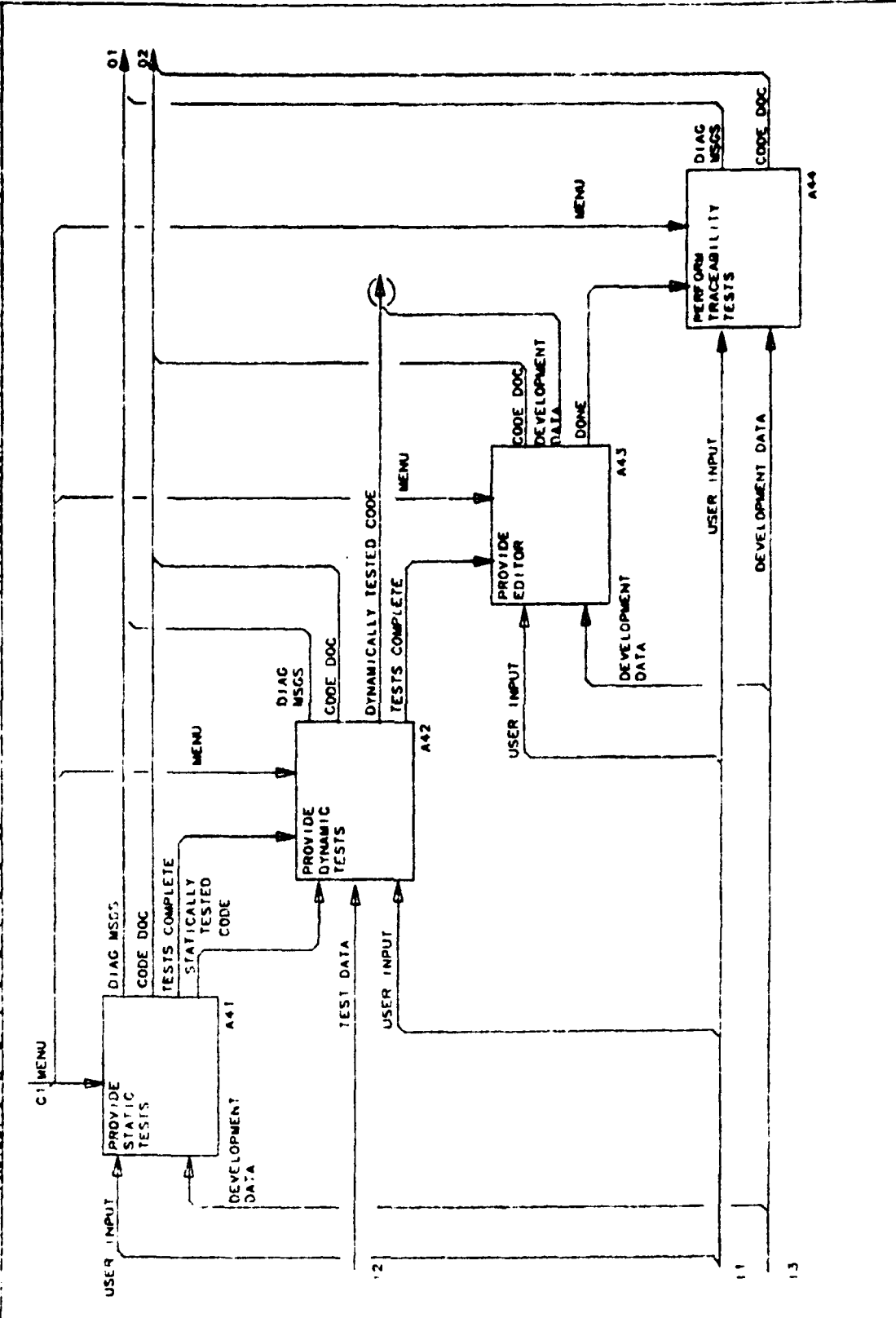
The exception is necessary because of the additional information provided with the detailed design. This information is the pseudocode algorithms which are included as a part of the detailed design data dictionary. If these algorithms are written in "executable" pseudocode, an example is PDL, then they may be run to test interfaces, nesting structure, and keywords (8). Execution is quoted because it refers more to code interpretation and symbolic execution rather than program execution.

[illegible]

### A36 Begin Execution Tests

During the detailed design the development information is refined to include pseudocode algorithms (e.g. PDL). These algorithms may be tested by execution. This process, Begin Execution Tests, is the first step in testing the "program" by execution; the "program" is created by concatenating the pseudocode algorithms according to the structure defined already defined. Editions on the "program" can be done by the user before the "program" is "executed". At this level of development Begin Execution Tests can test for interface consistency, properly nested constructs, data type consistency, and cross-referencing of keywords.

USED AT:	AUTHORITY: Keith A. Chomper		DATE: 10/31/84		WORKING		READER		DATE		CONTENT:	
PROJECT: SDUTN			REV: 1		DIRBT							
NOTES: 1 2 3 4 5 6 7 8 9 10					RECOMMENDED							
					X PUBLICATION							



MODE: SDUTN/10/A4.1	TITLE: Perform Code Tests	NUMBER:
---------------------	---------------------------	---------

#### A4 Perform Code Tests

The process of coding and debugging is perhaps the most understood of all the development phases, because it is the one phase that is common to all software engineers. There are numerous automated/interactive tools provided for this phase of development, and they can generally be categorized as either static or dynamic testing tools. Another tool necessary for code testing is the editor. This tool provides the capability to initiate the changes which will inevitably occur during debugging.

The activity most often neglected during the coding phase is documentation. This is unfortunate, because in an automated interactive environment the opportunity to make a "quick fix" can subtly lead the software engineer away from disciplined programming. Coding in the "quick fix" mode also leaves a gap in the development documentation from the "almost there" solution to the final solution. This is why automated/interactive documentation support is included with the traditional coding activities of testing and editing.



[illegible]

NOTE	CUM TO AS, I	TITLE: Perform Integration Tests	NUMBER

#### A5 Perform Integration Tests

During integration much of the same type of testing is required as in the coding phase. The differences in the tests which are used in this phase are the types of static and dynamic tests which are used (these differences are apparent at a lower level of design). The Test Methodology supports the static and dynamic test tools which are useful in the coding, integration, and maintenance phases.



#### A6 Perform Operations and Maintenance Tests

During operations and maintenance much of the same type of testing is required as in the coding phase. The differences in the tests which are used in this phase are the types of static and dynamic tests which are used (these differences are apparent at a lower level of design). The Test Methodology supports the static and dynamic test tools which are useful in the coding, integration, and maintenance phases.

APPENDIX B  
Preliminary Design

## Appendix B

### SDW Test Methodology Preliminary Design

#### Introduction

The structural model is presented twice in this investigation. It is presented as the Preliminary Design in this appendix, and then it is refined and presented again as the Detailed Design in Appendix C. The Preliminary Design is a high-level logical design which outlines the structure of to SDW Test Methodology.

Structure charts are used to represent the Preliminary Design; justification for choosing the structure chart representation is given in Chapter Three. The purpose of the structure chart model is threefold:

- 1) to present the SDW Test Methodology in a structured modular fashion,
- 2) to increase insight of each function's purpose, as it applies to the whole, beyond that given by the Requirements Model, and
- 3) to represent a design which is implementation feasible and yet still reflects the needs of the user.

Point One is satisfied through the discipline enforced by top-down design. Point Two is satisfied by the additional detail provided in the structure chart representation. The satisfiability of Point Three is arguable at this lifecycle phase, but the conviction of this investigation is that

Point Three is satisfied by the Preliminary Design.

#### Discussion on the Preliminary Design

One of the characteristics of lifecycle development is progressive refinement by iteration. Iteration allows the software engineer to refine the software by successive re-examinations of the development information. The repetition of the iterative process helps improve the software engineer's understanding of the problem and the solution; during this process he sees the development in a variety of different analysis and design formats. Iteration also helps him to ferret out errors in the analysis, design, and code.

The Preliminary Design is one of the products of the preliminary design phase. It has passed through this iterative process; consequently, it has been refined and modified. During the preliminary design phase new ideas concerning the methodology were formed, improvements in the implementation were discovered, and certain restrictions on the implementation became apparent. None of the ideas are revolutionizing, the improvements are modest, and the restrictions minor, but all of these events influenced the design in some way. They are presented below with a discussion on the parts of the design each one affected.

The Ideas. While drafting the Preliminary Design it became apparent that the iterative process in software development was both purposeful and beneficial to the software engineer. It is helpful in advancing the software engineer's understanding of the project by creating a setting for in depth analysis and design. Therefore, the Test Methodology should promote and assist the iterative process. Care is taken not to over-automate testing (if this is possible) to the point where the software engineer no longer interacts in the iterative process of analysis, design, and implementation. Rather, the Test Methodology seeks to manage lifecycle testing and to provide automated testing and documentation support to aid the software engineer in determining the software's quality. The Test Methodology is a team approach, to testing integrating the consistency and rigor of the automated tests with the judgement and intelligence of the software engineer.

An example of the effect which this ideology has on the design is Module 2.5.3, Compare Functional Intent. This module was originally imagined as providing a simple comparison of function names in the requirements and design phases to test for functional completeness. During the preliminary design phase it became apparent that by adding the intelligence of the software engineer to comprehend the function definitions the test would be more accurate.



Another example is in Module 2.4.2.2. In this instance the software interacts with the user to discover what relationships exist between data of adjacent phases; the user's job is to find the obscure relationships which the unmatched entries might possess.

The Changes. The changes that the design underwent are minor and many relate to the ideas expressed in the proceeding section. Other changes are the control flags in Modules 1.1, 1.2 and 1.3 which are added to determine the phase in which the Methodology is operating in. SADT Modules A131 and A132 were combined (now Module 1.3.1), because of the relations defined in the development database which link these pieces of information together. Finally, changes were made in the input/output data flow of the structure charts to reflect the evolving understanding of how the data is passed between the modules. These changes have since been incorporated in the Requirements Model also.

The Restrictions. A restriction which is increasingly important in the Test Methodology's design is its need to interface with the Data Dictionary Generation Tool currently under development at AFIT (58). While this restriction does not limit the scope of the Test Methodology, it does affect the design, because it defines what development information is accessible to the Methodology's test tools through the relational development database.

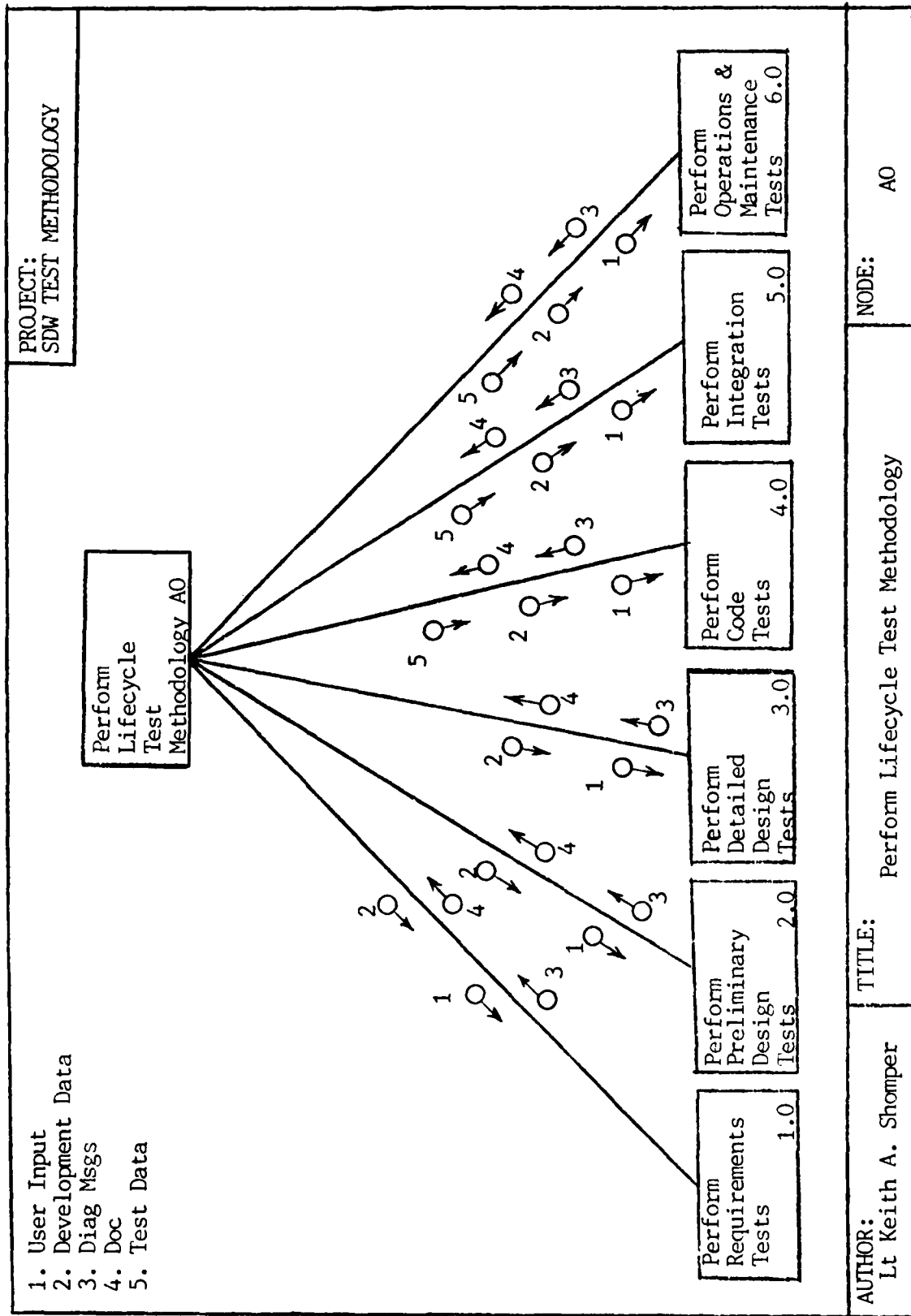
Another restriction on the design is time. As a first cut, the SDW Test Methodology will probably require enhancements and improvements. However, the design of this Methodology must proceed to the detailed design phase so that it can meet the immediate demands of the AFIT software development community.

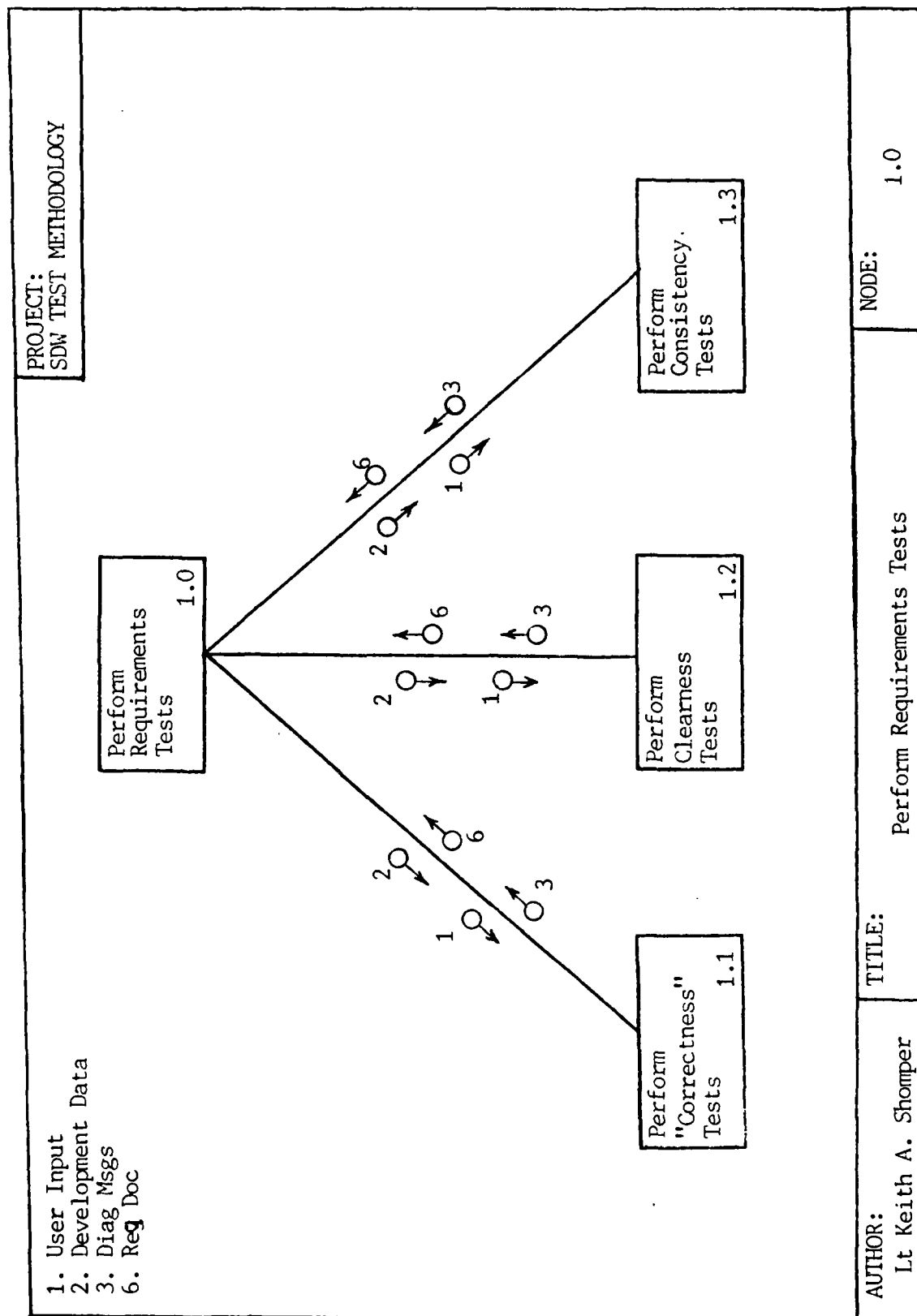
#### Summary

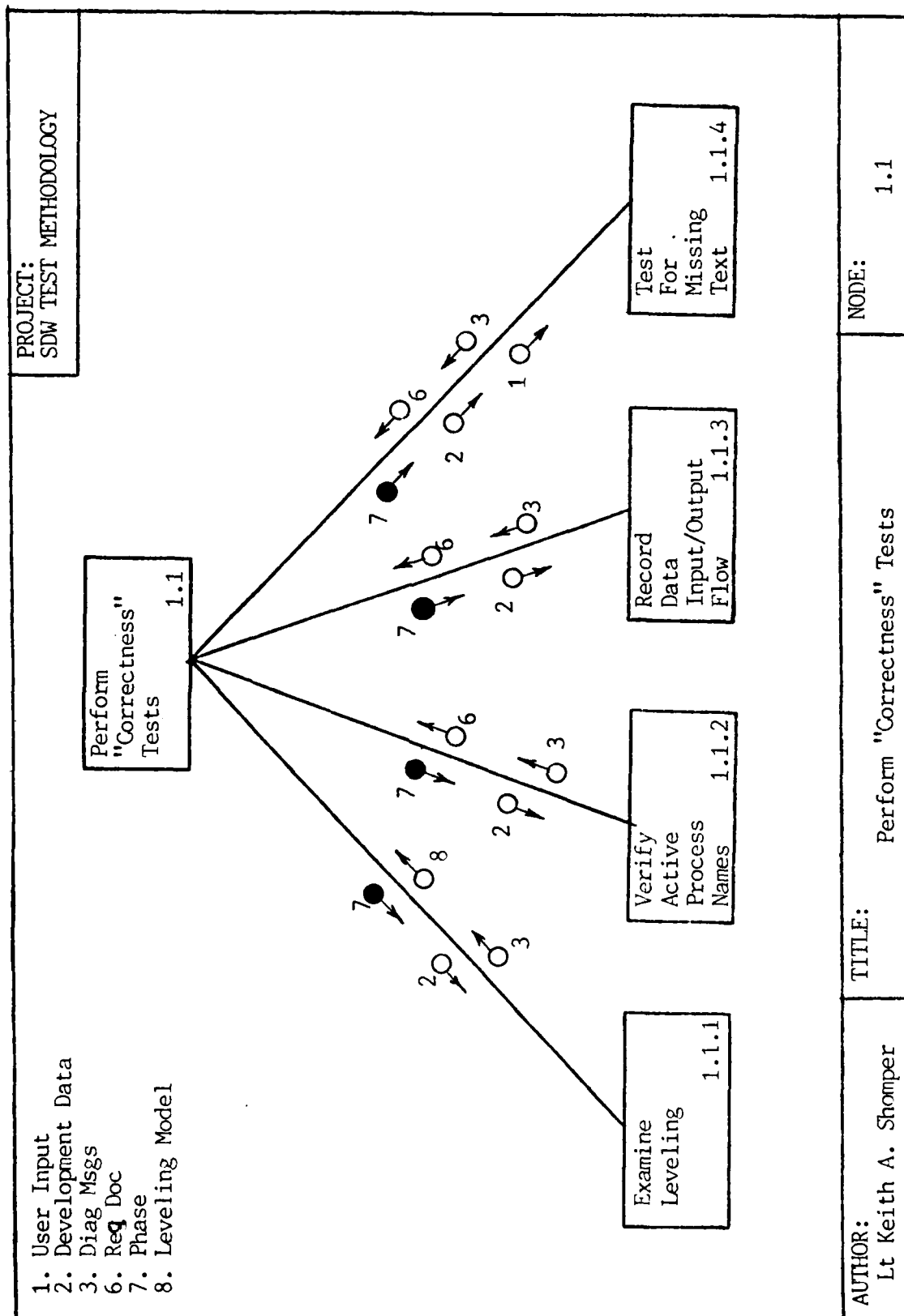
As the preliminary design phase progresses new ideas are formed, changes in user needs are discovered, and restrictions are confronted; these events have an impact on altering the design. The software engineer needs to consider these variables and present a design that reflects attention to them and satisfies the requirements outlined earlier in the development. As changes in the SDW Test Methodology's environment have occurred, the Preliminary Design has been modified to reflect these changes.

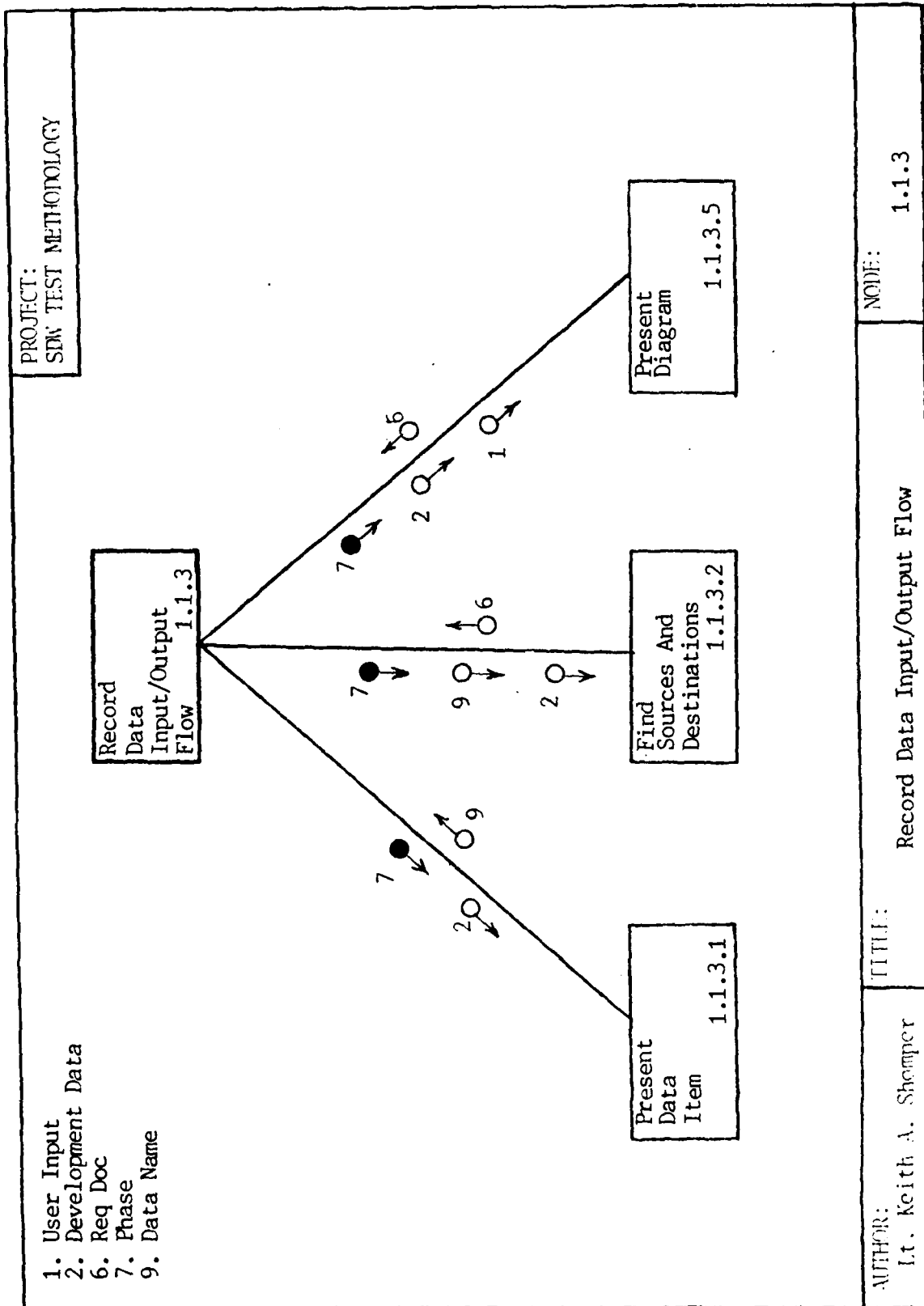
### Structure Chart List

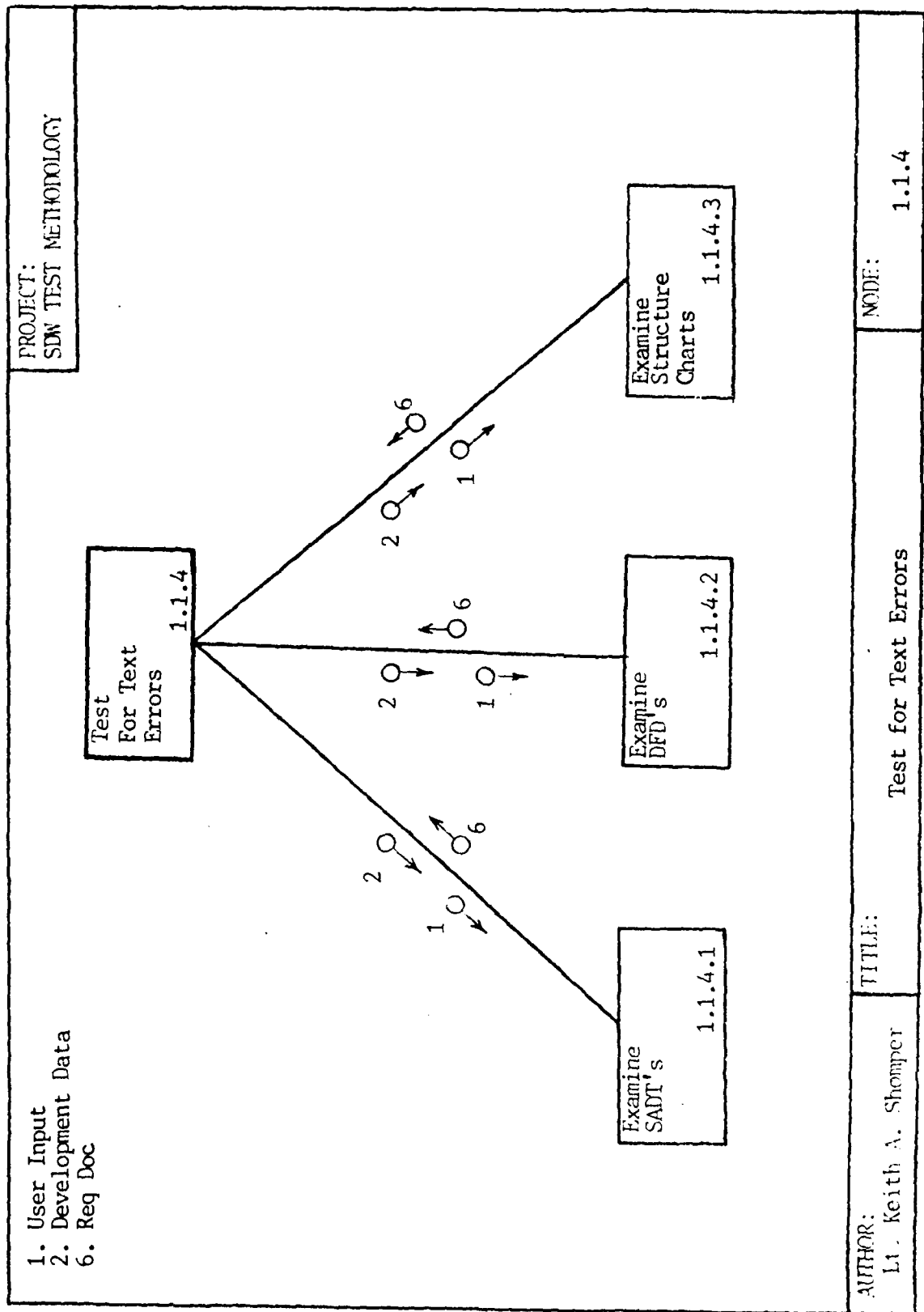
A0	Perform Lifecycle Test Methodology
1.0	Perform Requirements Tests
1.1	Perform "Correctness" Tests
1.1.3	Record Data Input/Output Flow
1.14	Test for Text Errors
1.2	Perform Clearness Tests
1.2.3	Check for Notational Abbreviations
1.3	Perform Consistency Tests
2.0	Perform Preliminary Design Tests
2.1	Perform Design "Correctness" Tests
2.2	Perform Design Clearness Tests
2.3	Perform Design Consistency Tests
2.4	Perform Design Traceability Tests
2.4.2	Verify Unmatched Entries
2.5	Perform Design Completeness Tests
2.5.3	Compare Functional Intent
3.0	Perform Detailed Design Tests
3.6	Begin Execution Tests
4.0	Perform Code Tests
4.4	Perform Code Traceability Tests
5.0	Perform Integration Tests
6.0	Perform Operations and Maintenance Tests













AD-A152 857

A TEST METHODOLOGY FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT(U) AIR FORCE INST OF  
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.

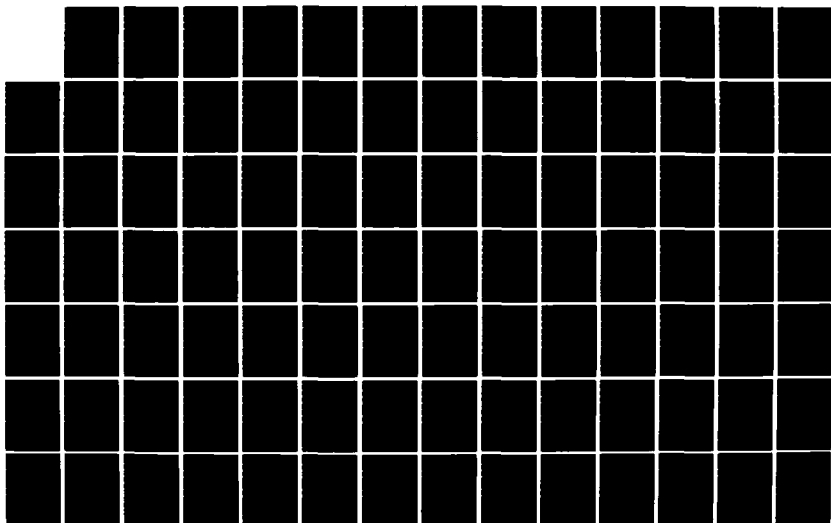
3/4

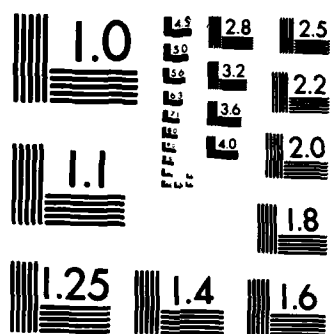
UNCLASSIFIED

K A SHOMPER DEC 84 AFIT/GCS/ENG/84D-26

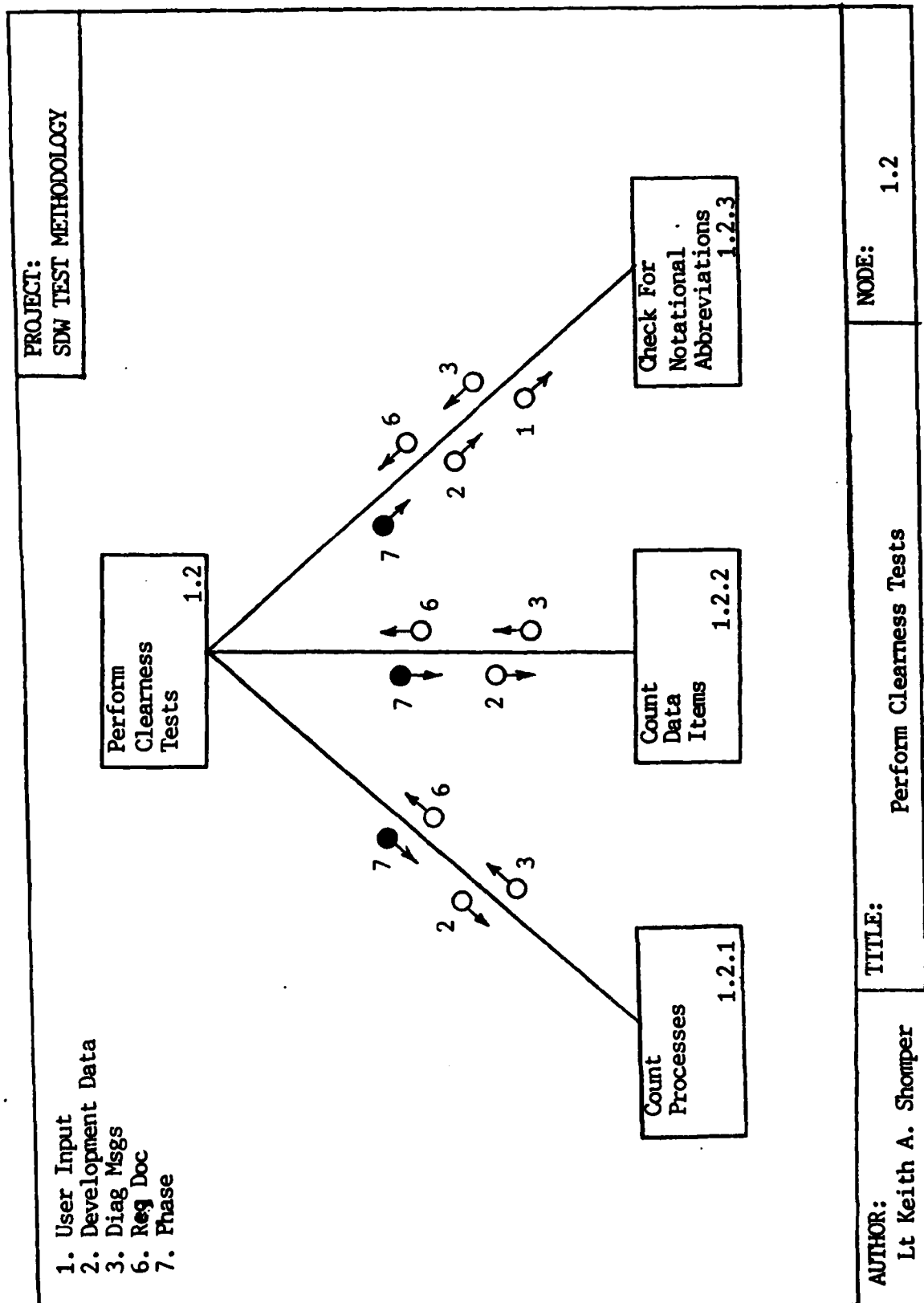
F/G 9/2

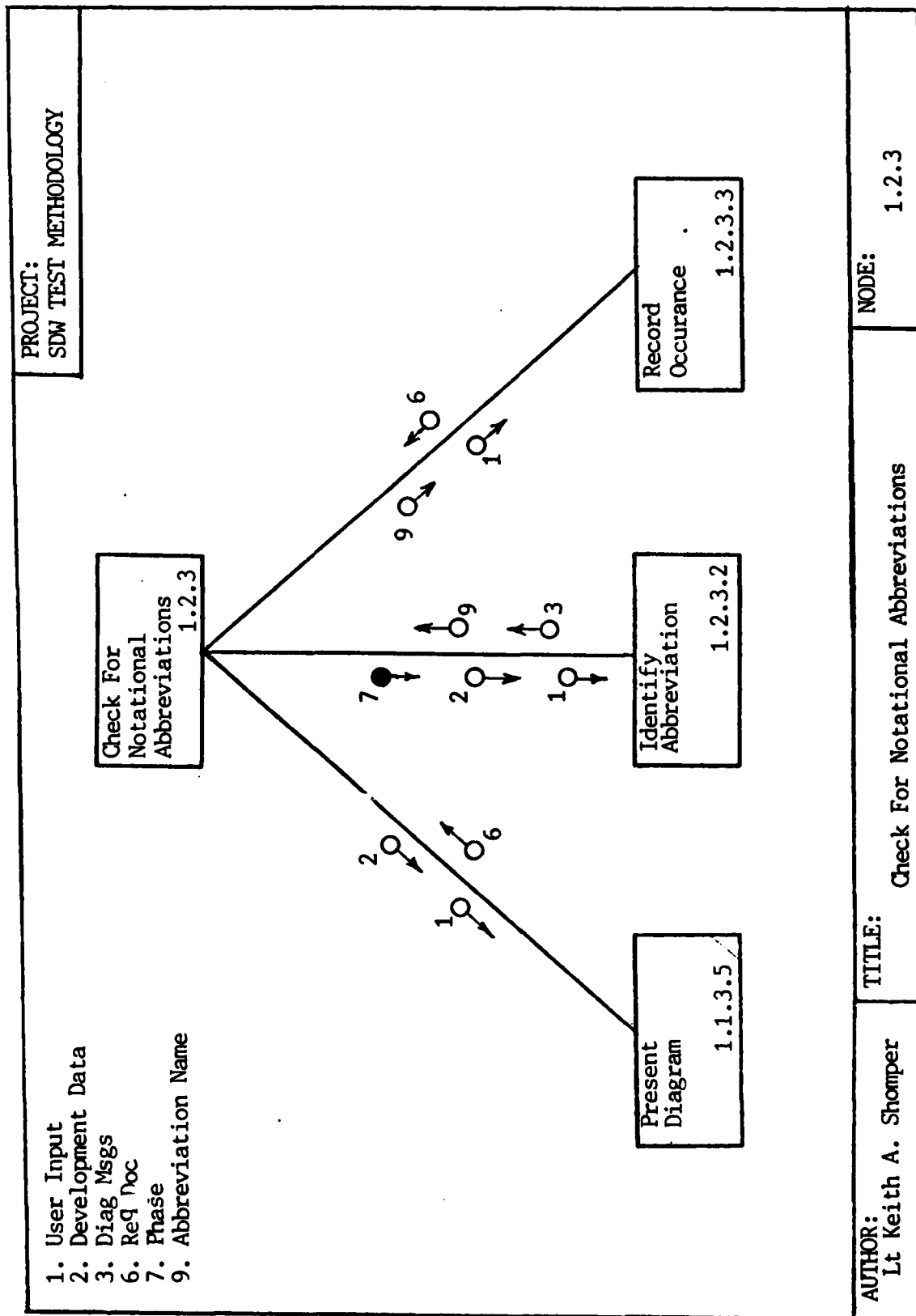
NL

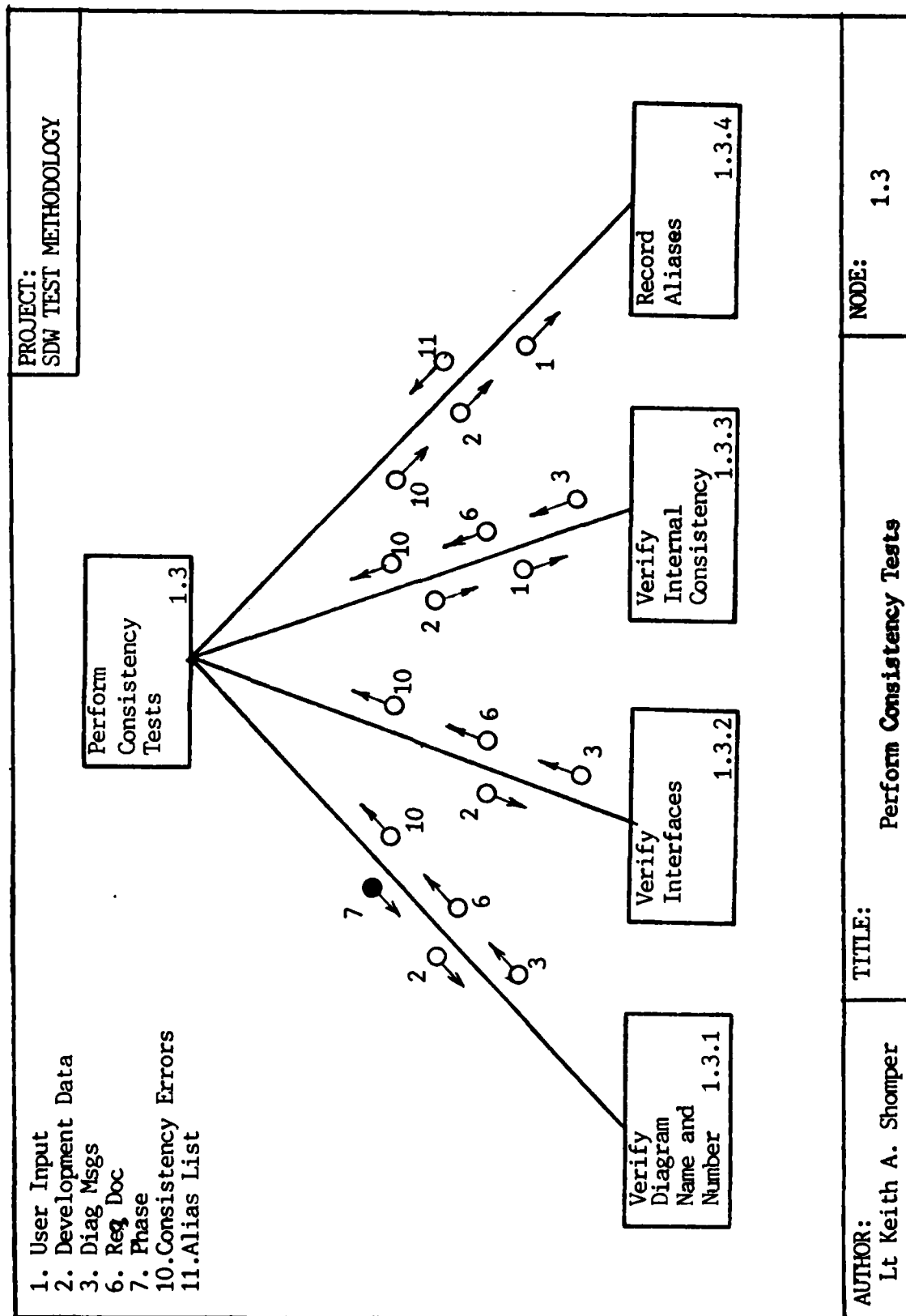


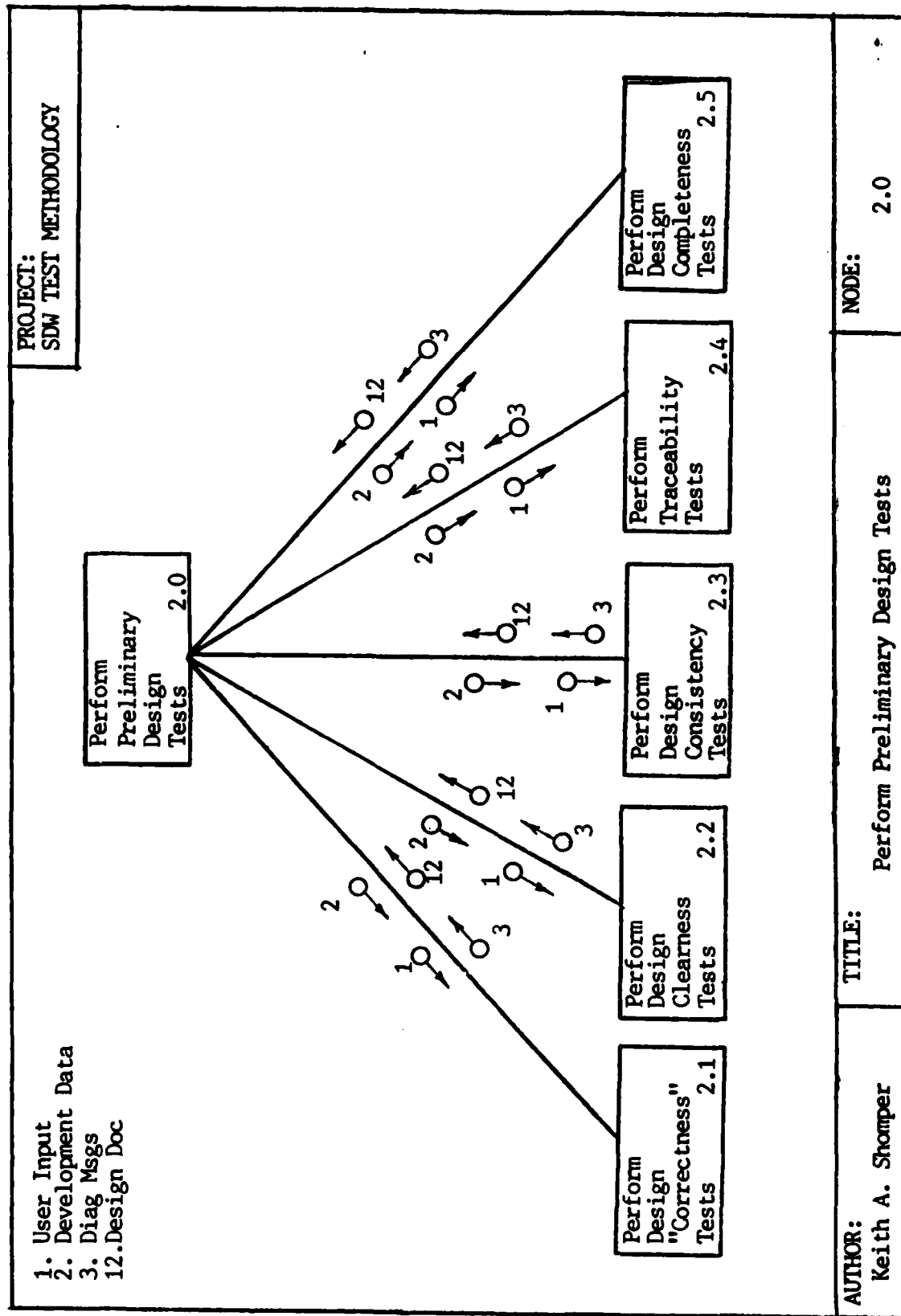


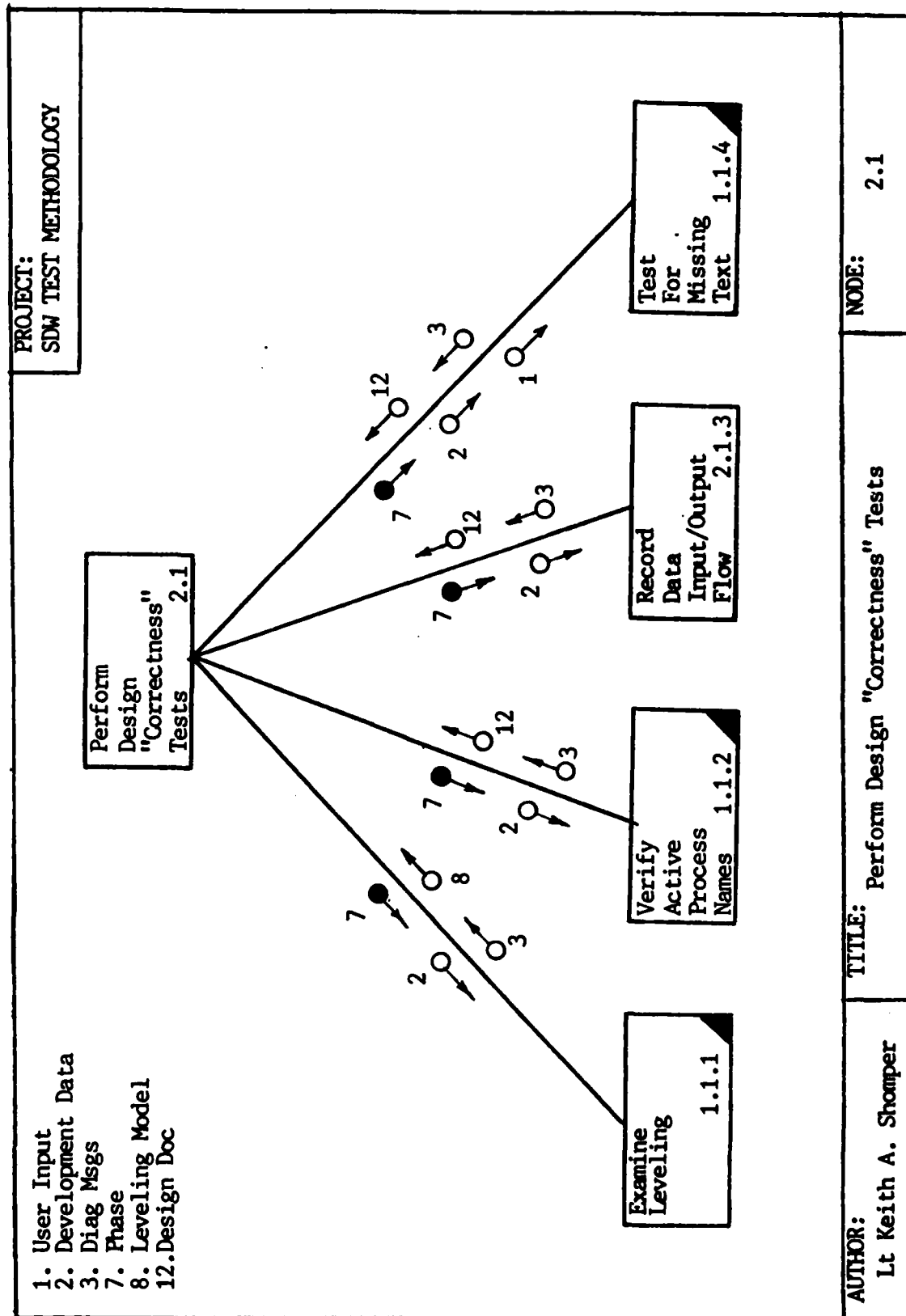
MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

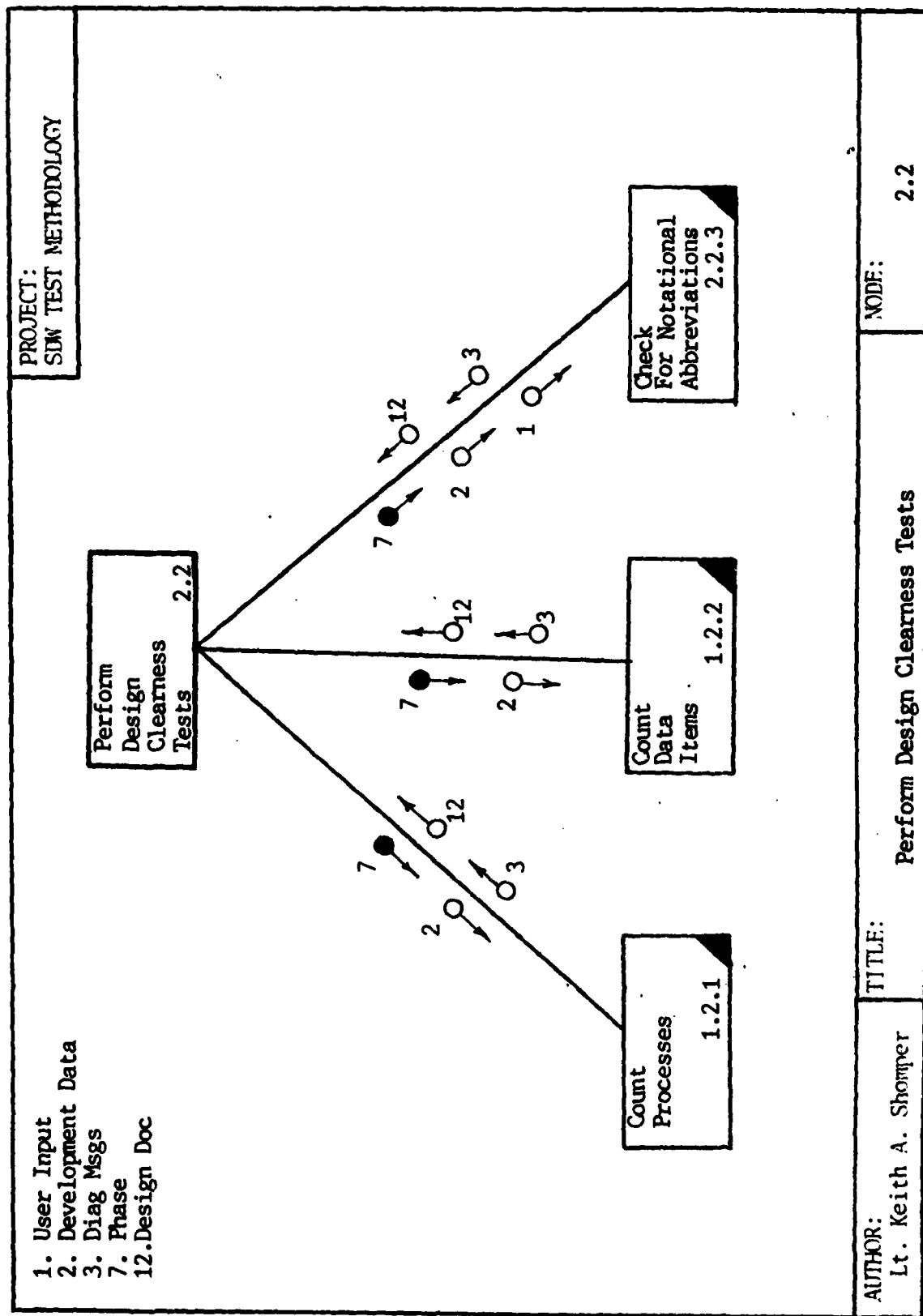




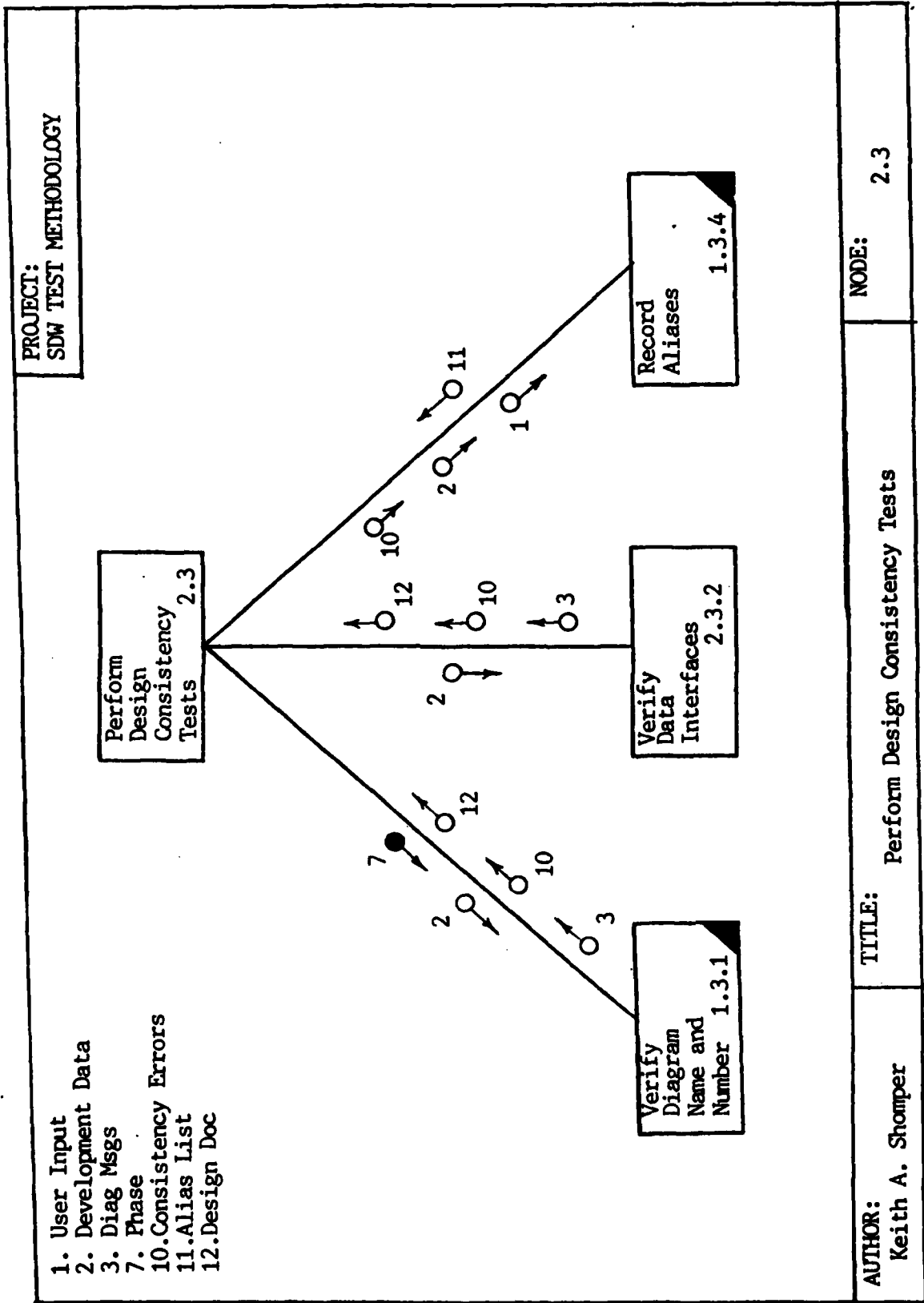




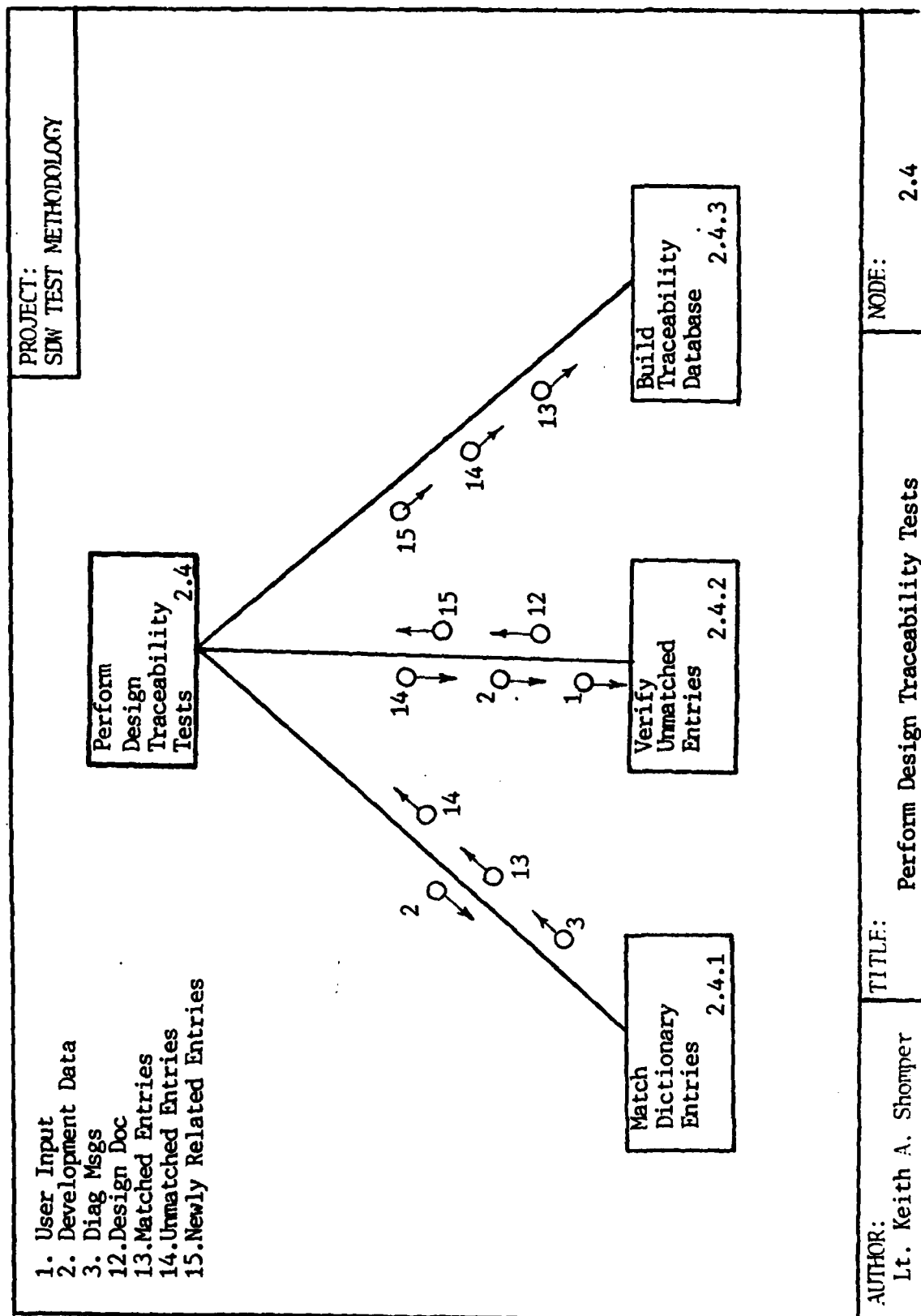


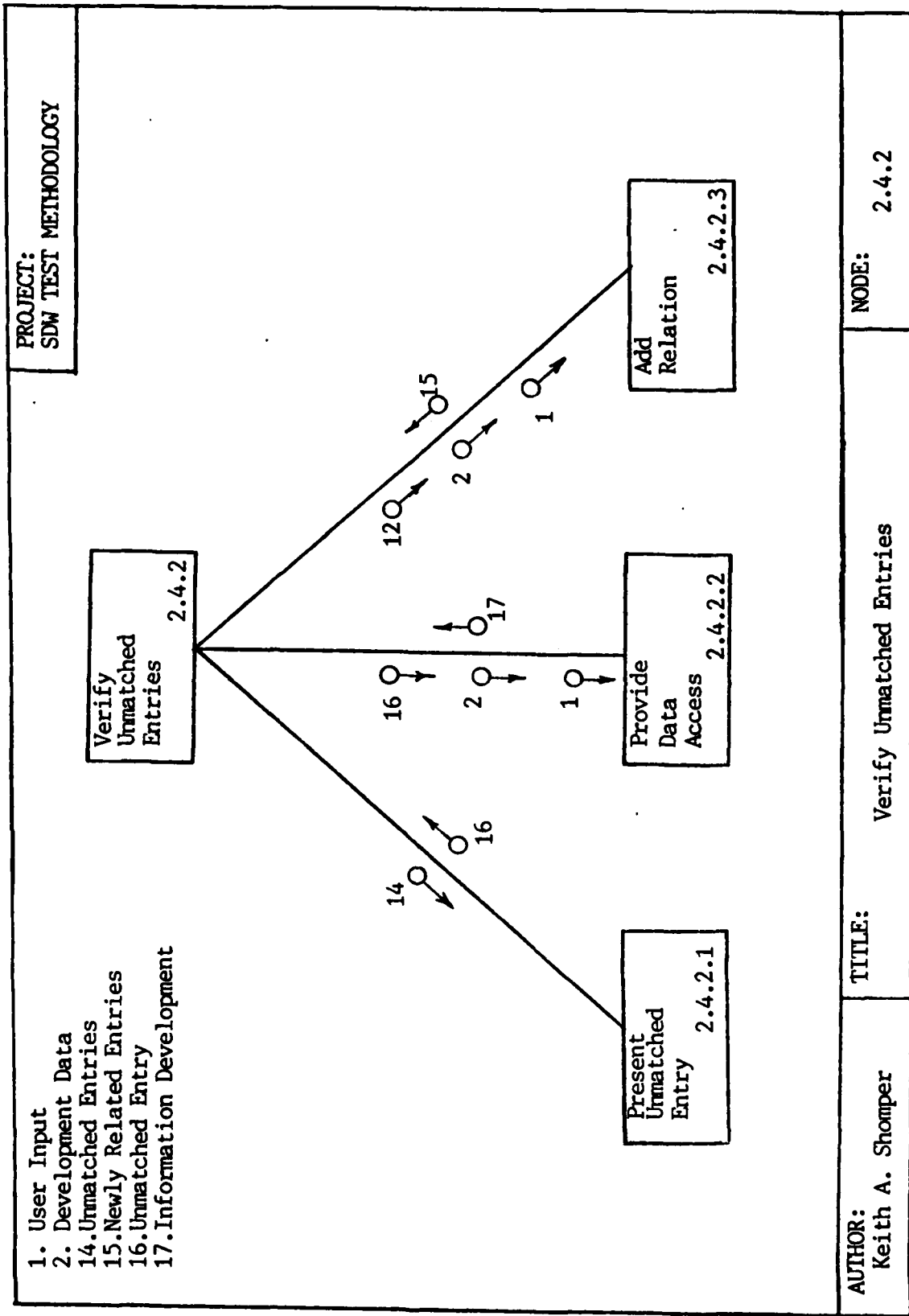


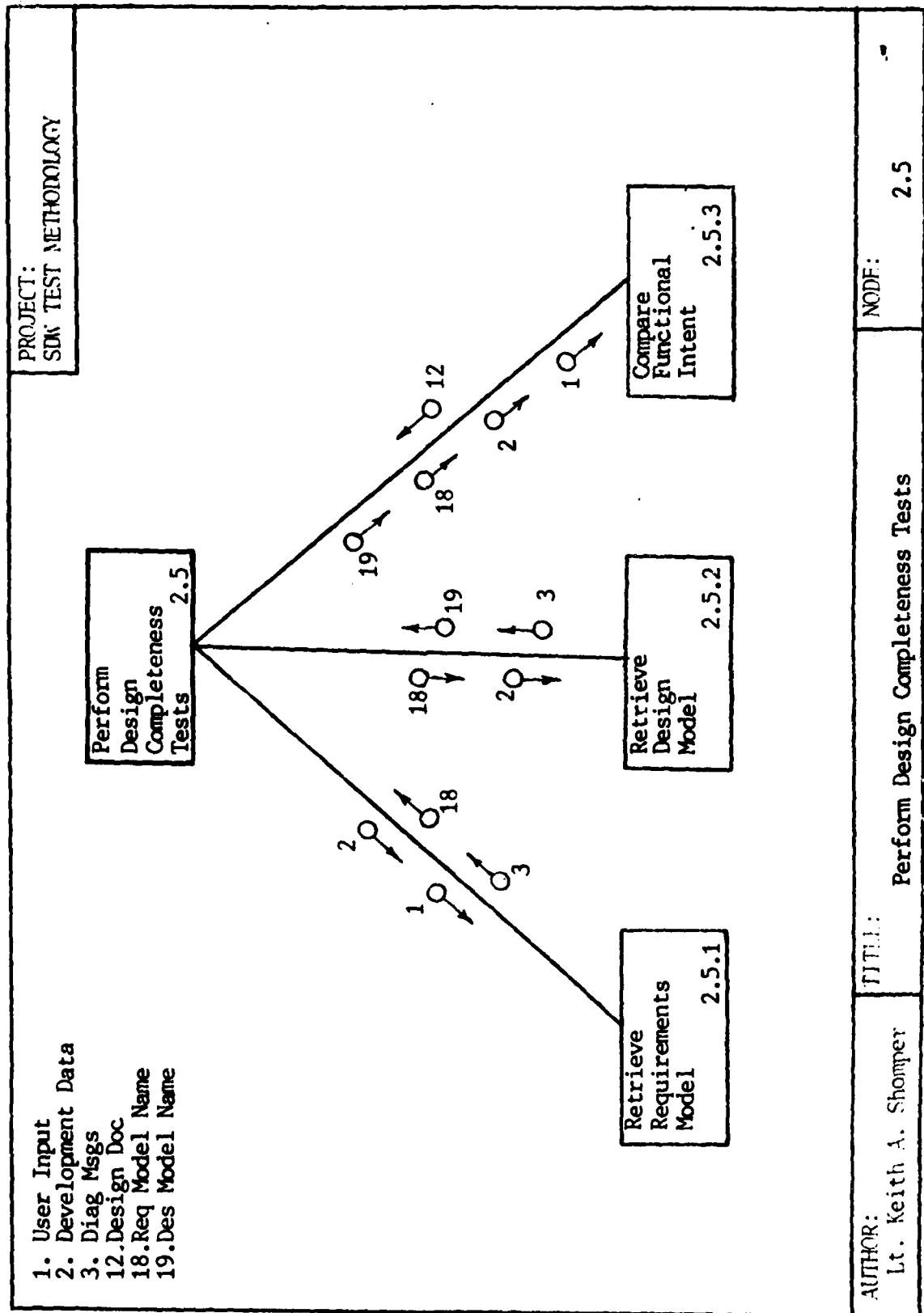


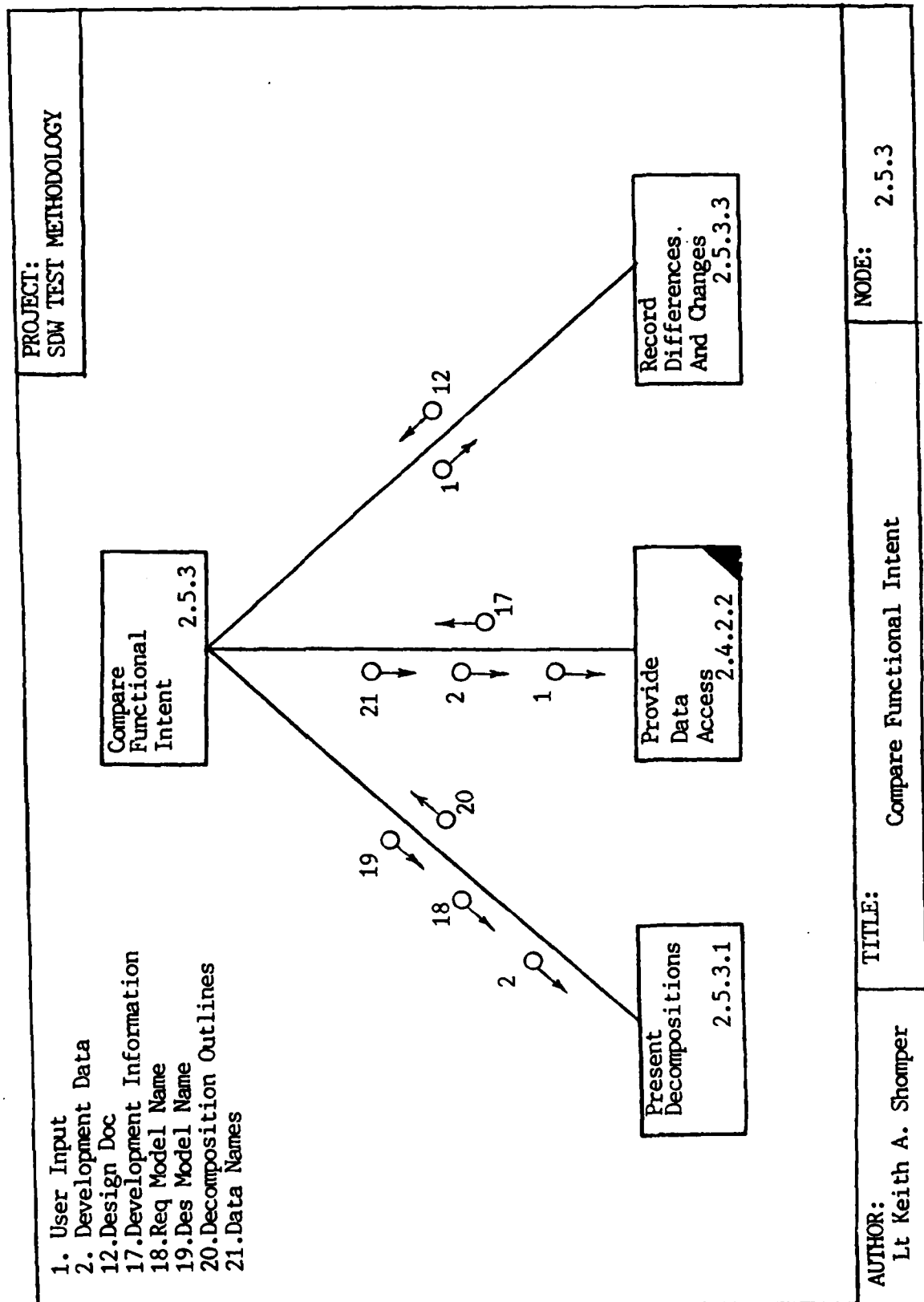


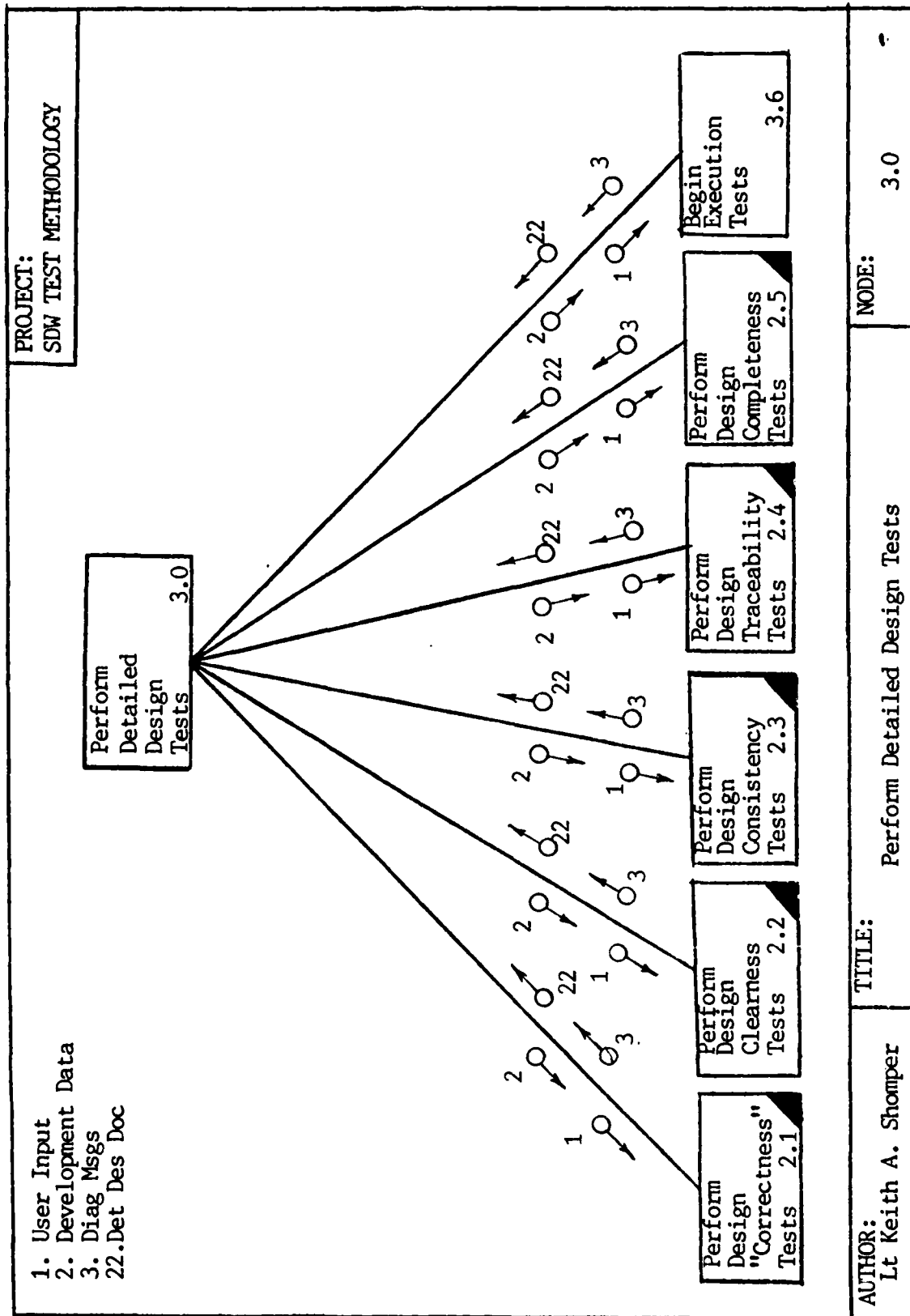
67

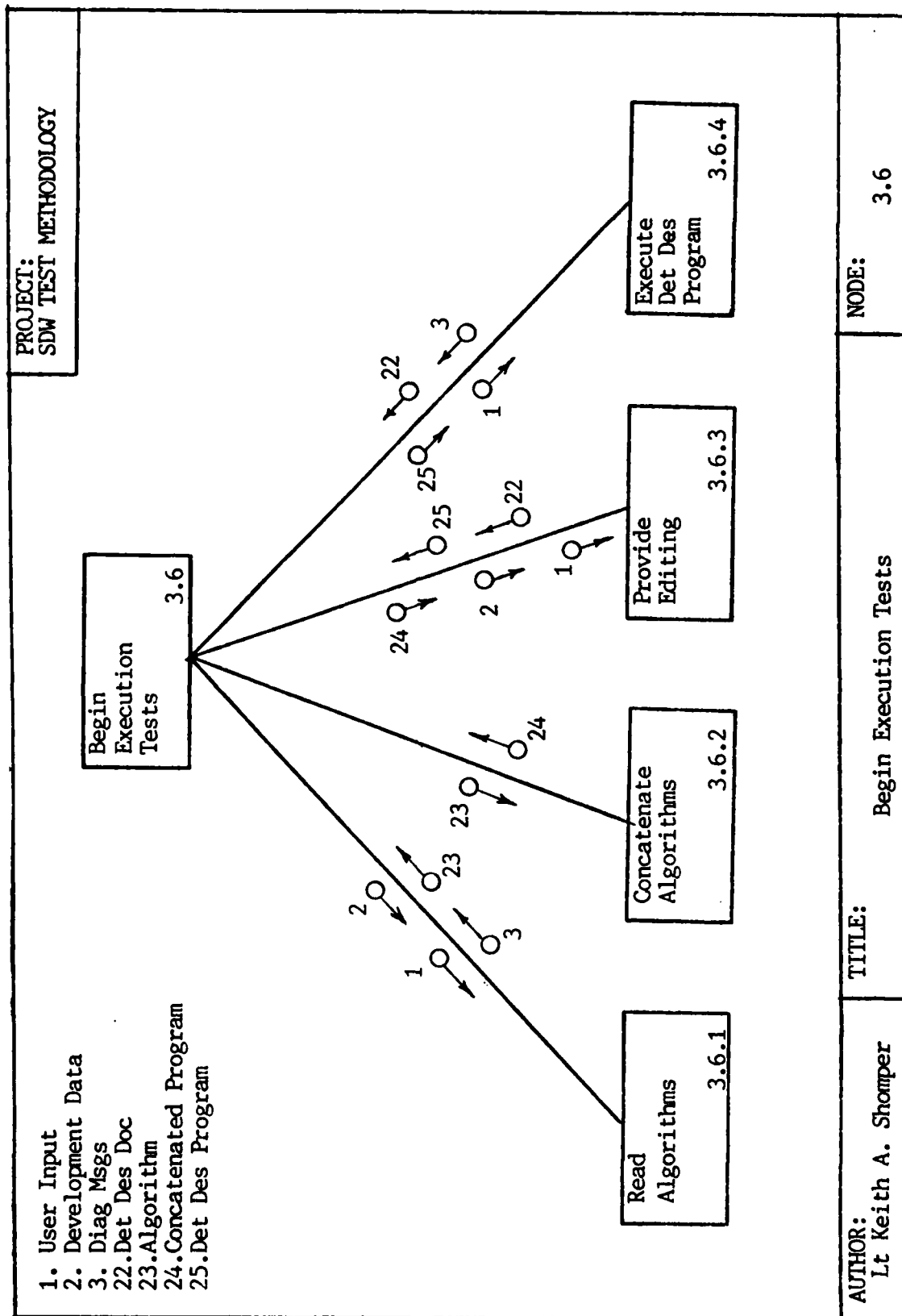


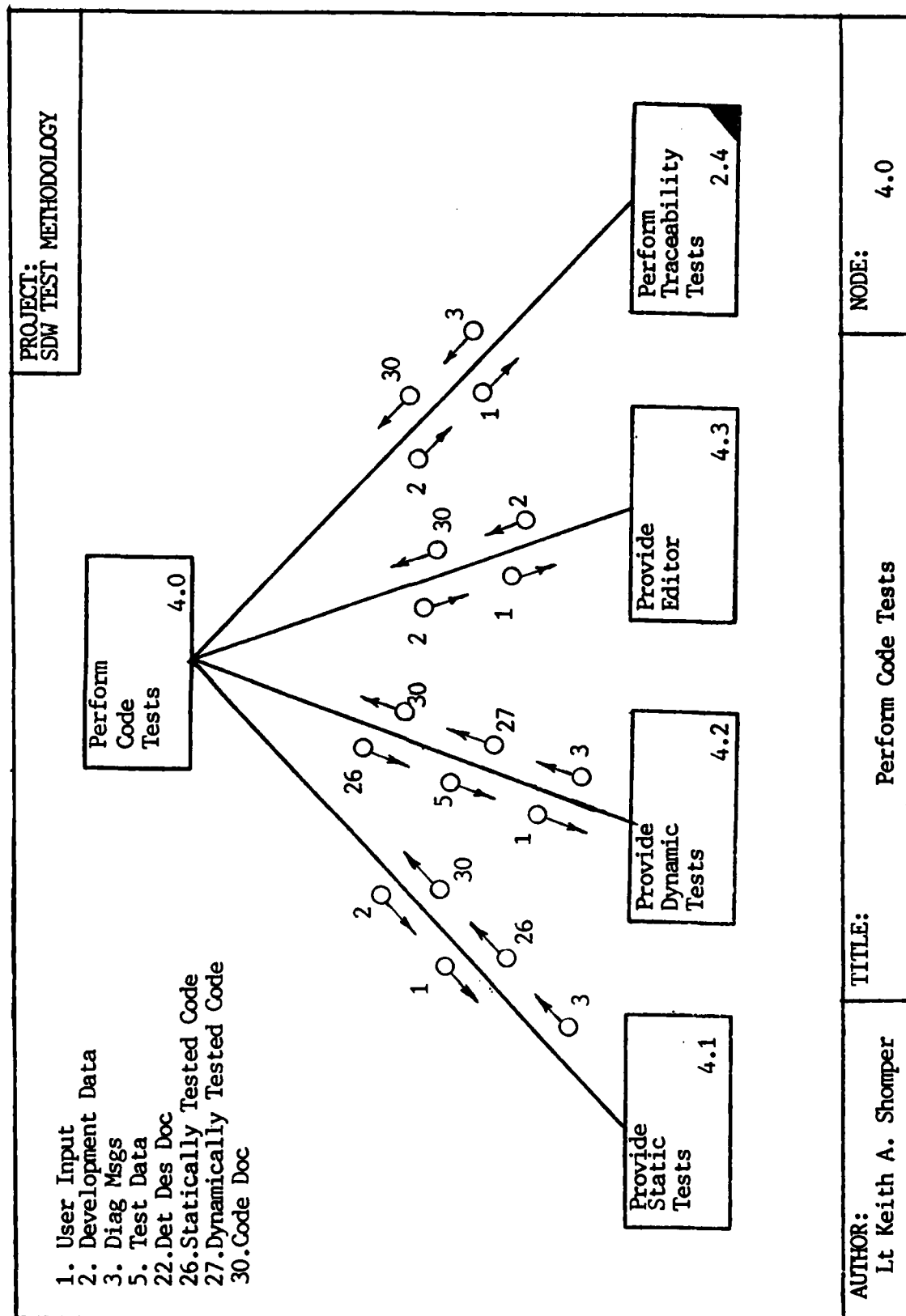




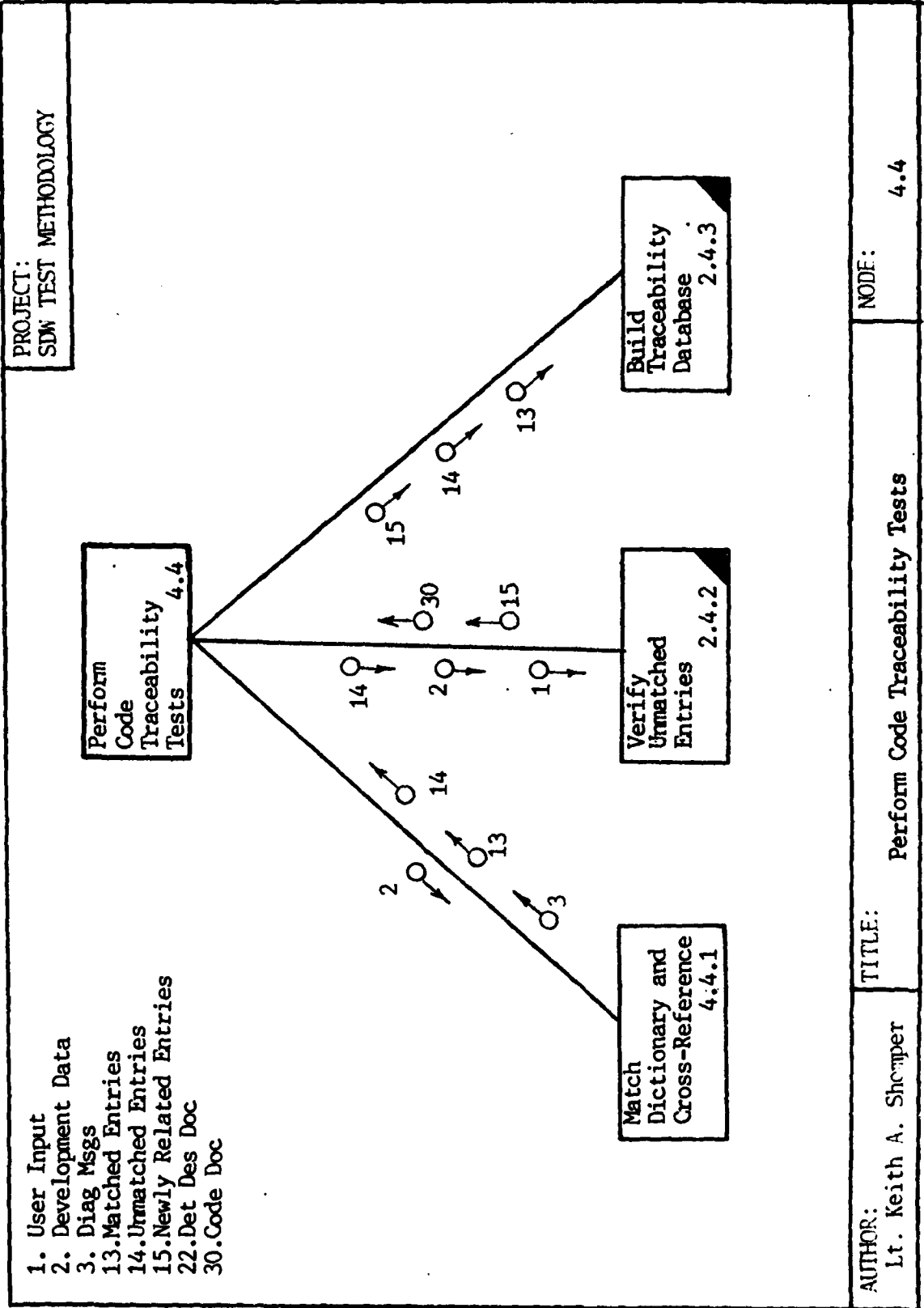


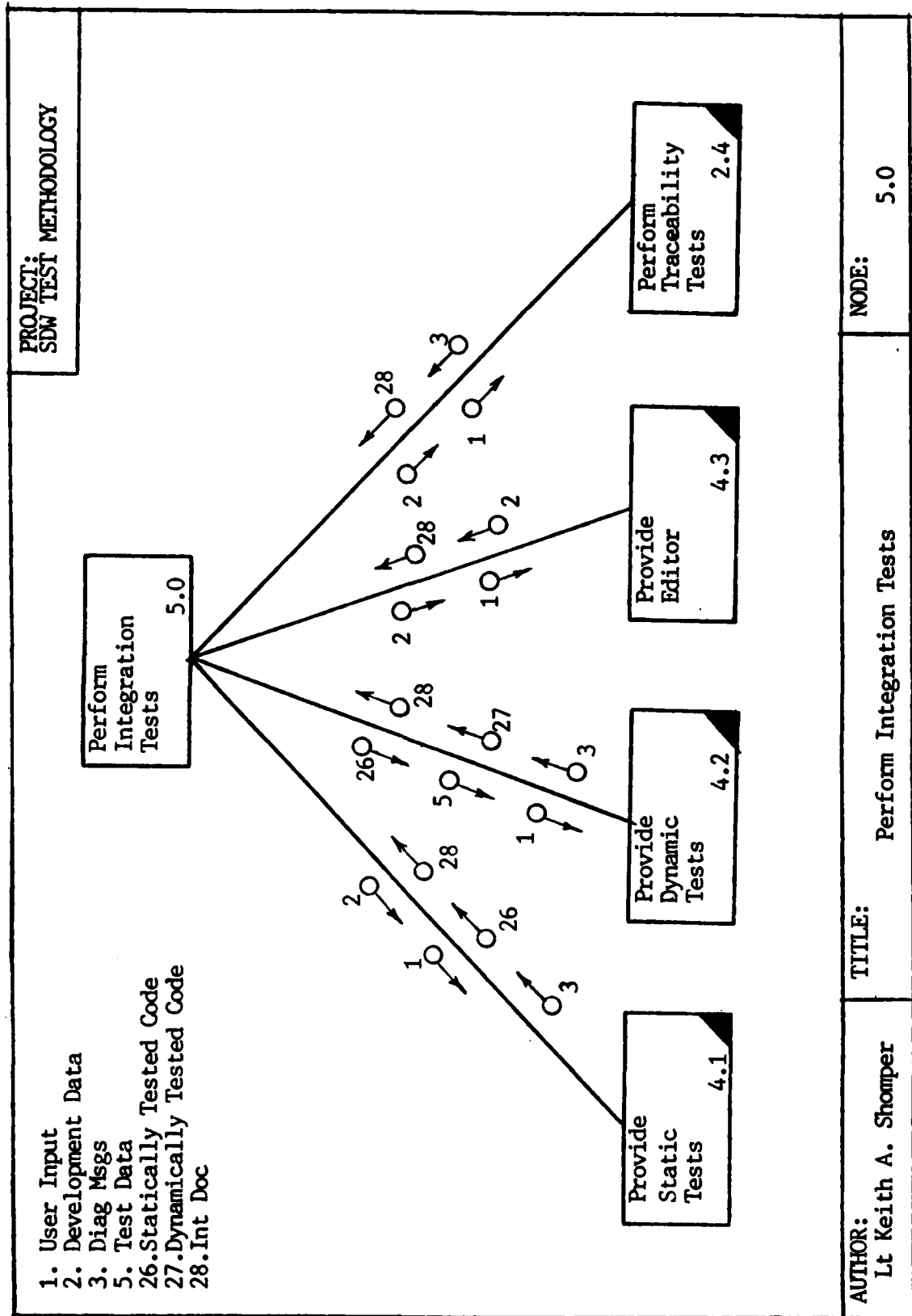


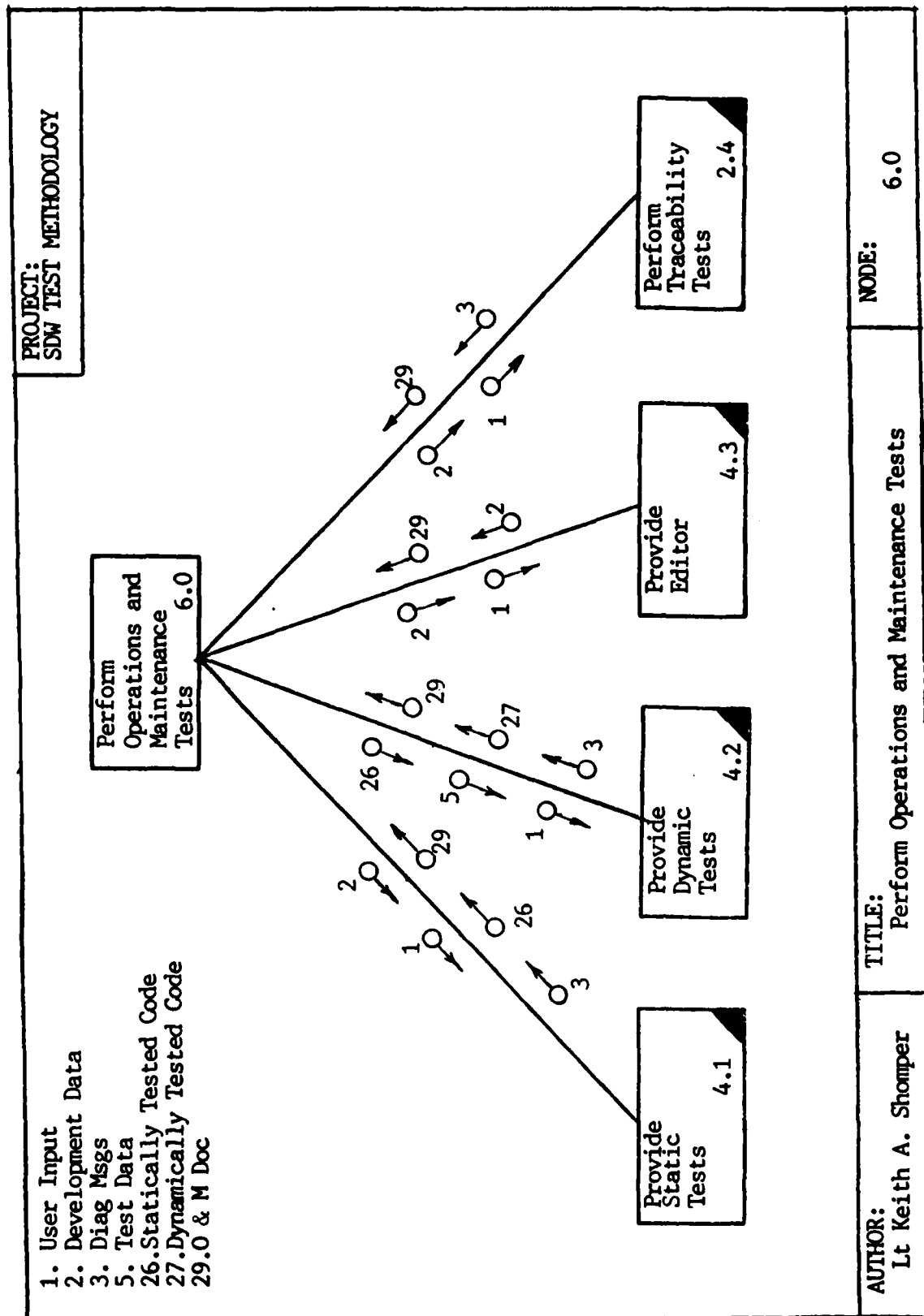












Appendix C  
Detailed Design

## Appendix C

### Detailed Design

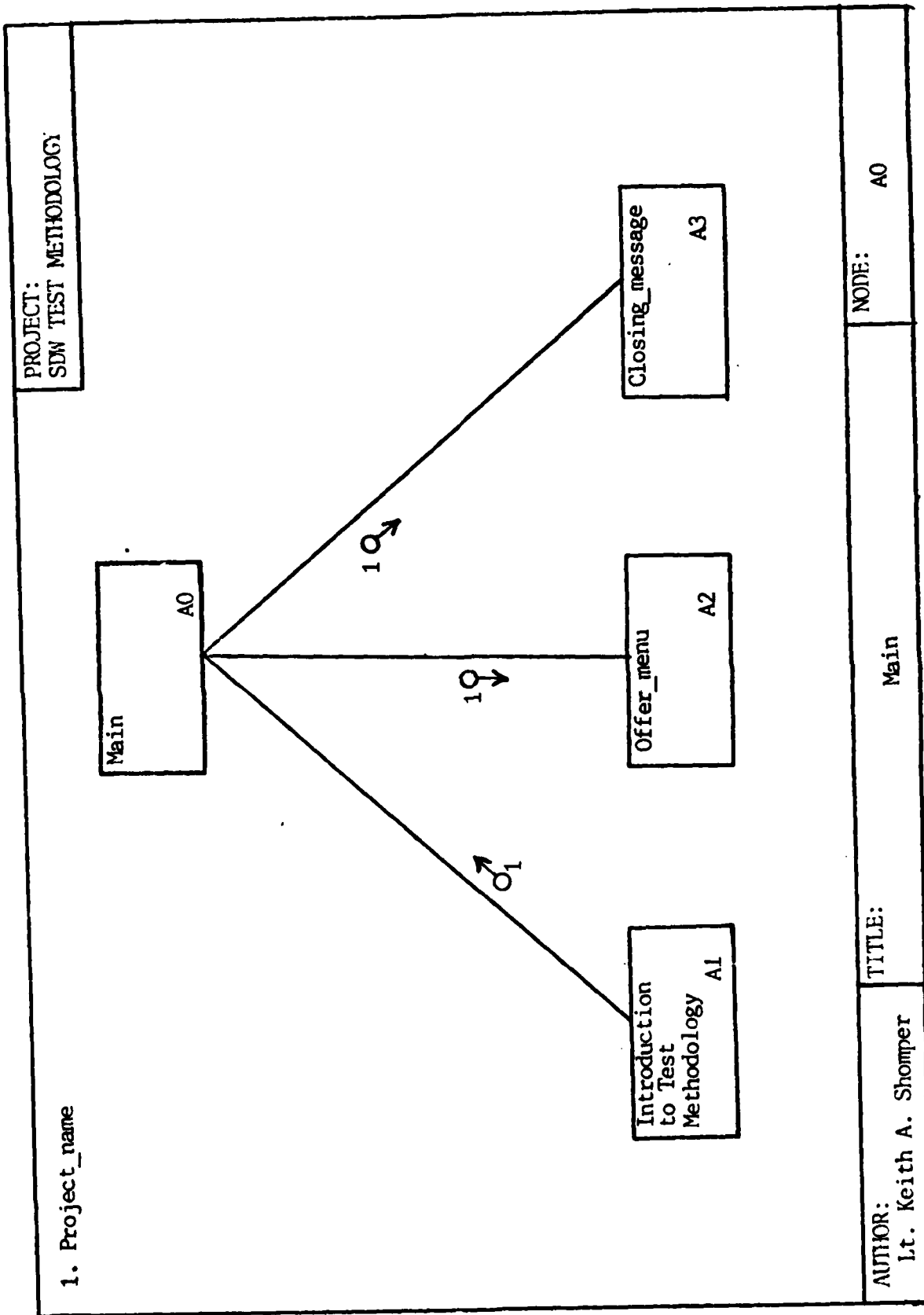
#### Introduction

In this investigation the Detailed Design is divided into two parts. The Structural Design and the Algorithmic Design. This division is primarily done to aid in the transition from the graphical representations employed in the requirements definition and preliminary design phases to the written code of the implementation phase.

This design is formatted by first presenting the structure chart model immediately followed by the pseudocode corresponding to that model. The pseudocode is written in the Program Design Language (8). The differences between this design and the Preliminary Design are quite significant, particularly in the upper lever of the Detailed Design. This is because a transition from the logical viewpoint to the implementation or physical viewpoint is made. Two major differences between the designs are noteworthy. The first is that the top-level logical structure of the Preliminary Design is now incorporated into the design for the menu structure (see figure 11, page IV-27). The second is that the data which is passed between modules has significantly changed. This change in data in the Detailed Design represents the actual expected module parameters as opposed to a logical conception of data flow represented in the Preliminary Design.

### Node List

A0	main
A2	offer menu
A26	execute command
A262	select
1.1.1	ver nam num
1.1.1.2	ver name
1.1.1.3	ver num
1.1.2	ver int
1.1.3	ver int cons
1.1.4	record aliases
1.2.1	exam level
1.2.1.2	build tree
1.2.1.3	print tree
1.2.2	ver act nam
1.2.3	rec data flow
1.2.3.4	det src dst
1.2.4	test txt err
1.3.1	cnt pro
1.3.2	cnt data
2.2.3	rec data flow
2.4	trace
2.4.2	mtch dic pro
2.4.3	mtch dic data
2.4.4	det asso umtch
2.5	complete
2.5.3	ret func info
2.5.4	comp int
4.2.2	int test



A0

Detailed Design  
Perform Lifecycle Test Methodology

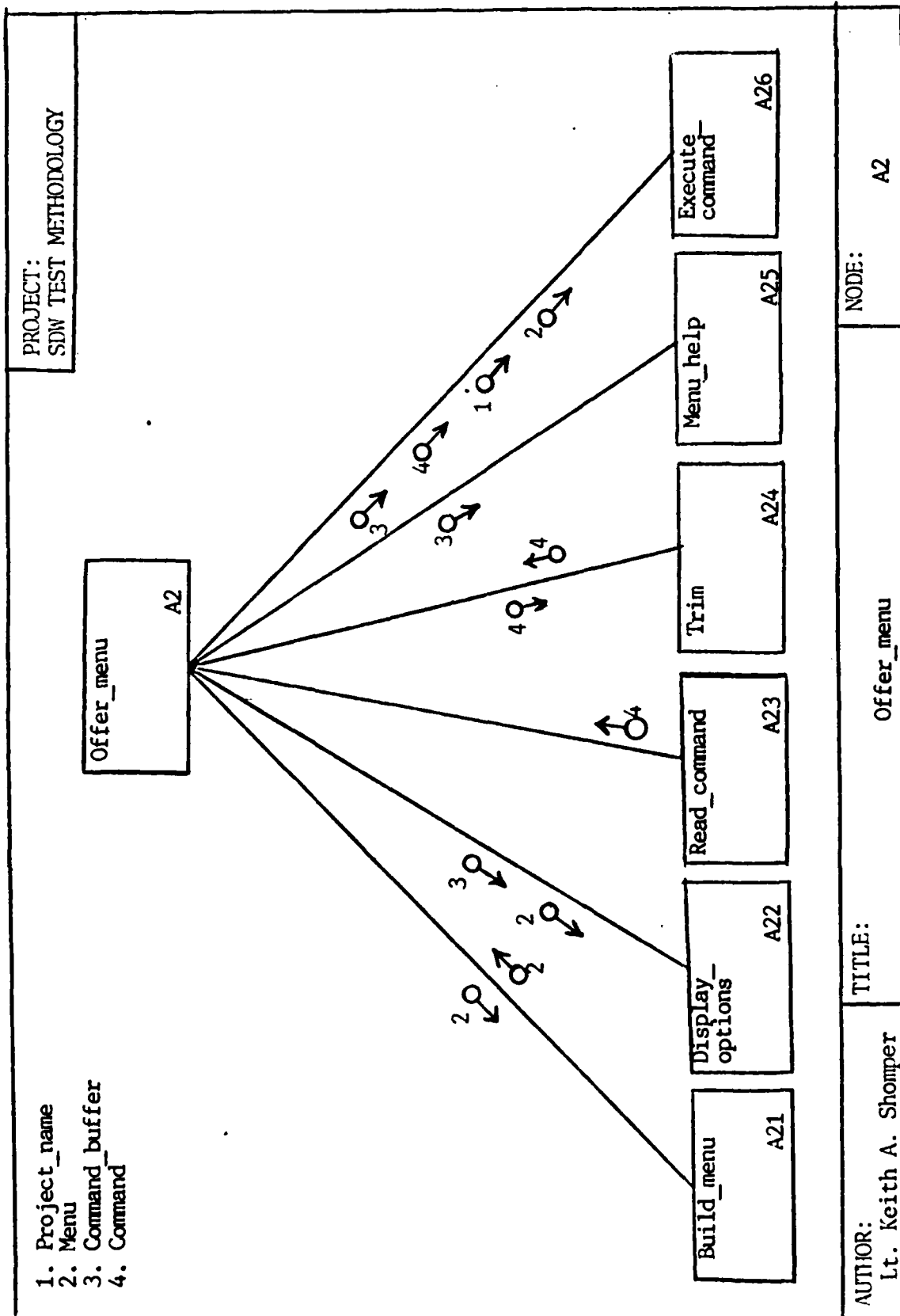
A0

main:

```
call introduction to test methodology
DO WHILE test methodology is still needed
    offer menu
    prompt user for quit command
ENDDO
call closing message
```

END main:





A2

Detailed Design  
Offer Menu

A2

offer menu:

    build menu

    DO WHILE the command to exit is given

        display the current menu options

        read user command

        IF the command is help

            provide the help facility

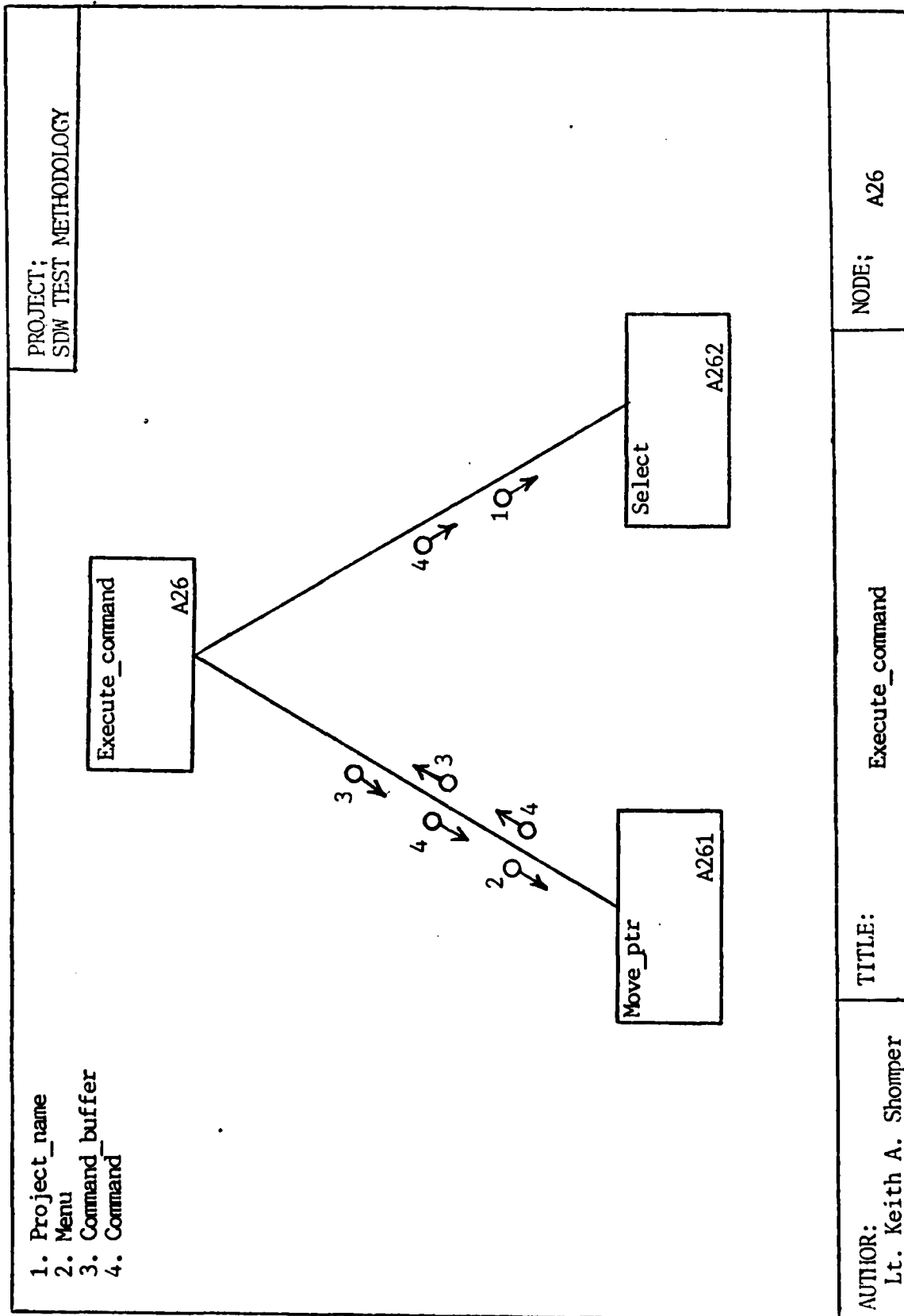
        ELSE

            execute the command

    ENDIF

    ENDDO

END offer menu:

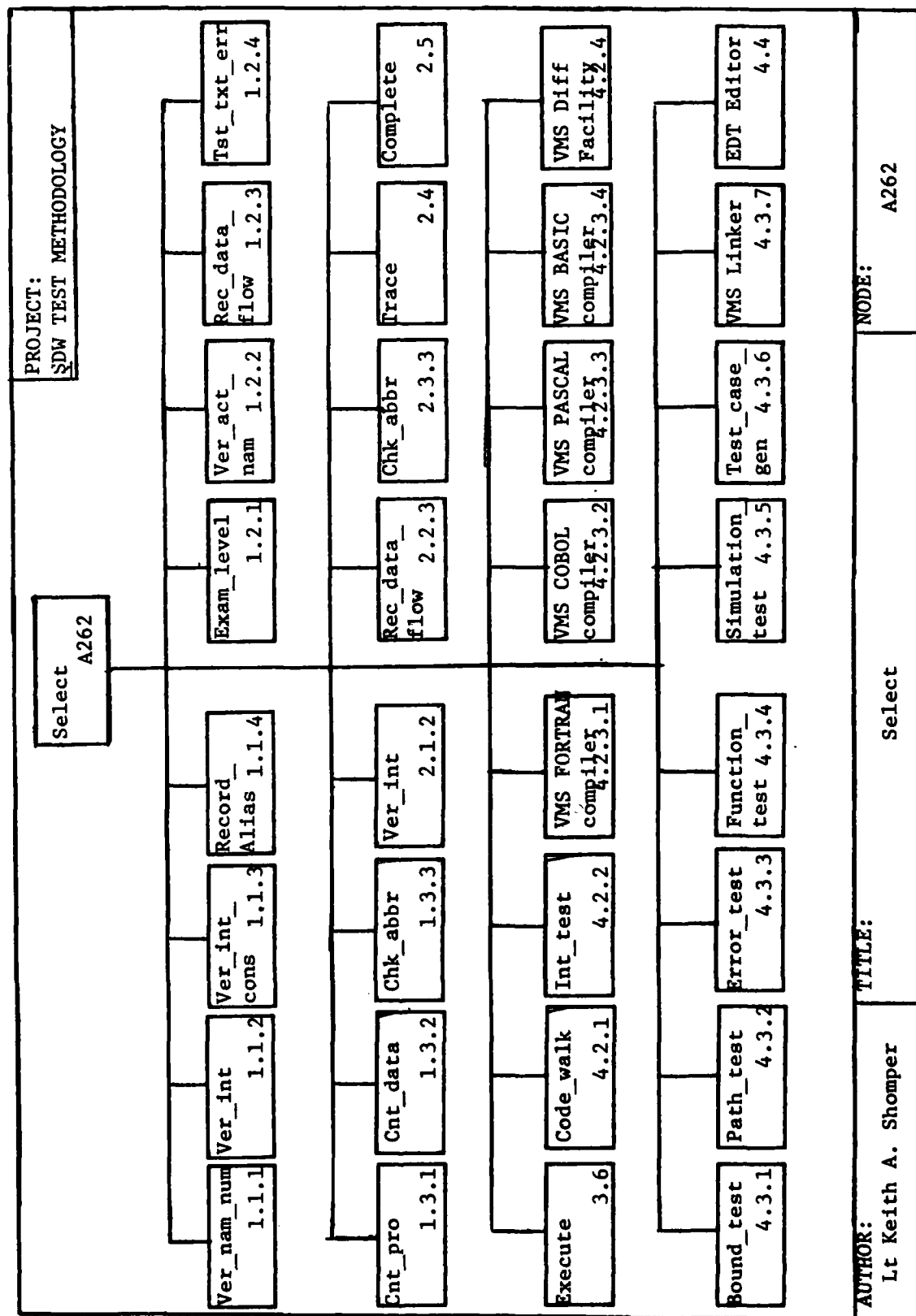


A25

Detailed Design  
Execute Command

A25

```
execute command:
    validate the command
    IF command is exit
        empty command word
        quit
    ELSE
        put command in command word
        move menu place marker to selected
option
        IF executable level is reached with
this command
            call select module
            move menu place marker up one menu level
            ENDIF
            record traceback path
        ENDIF
END execute command:
```



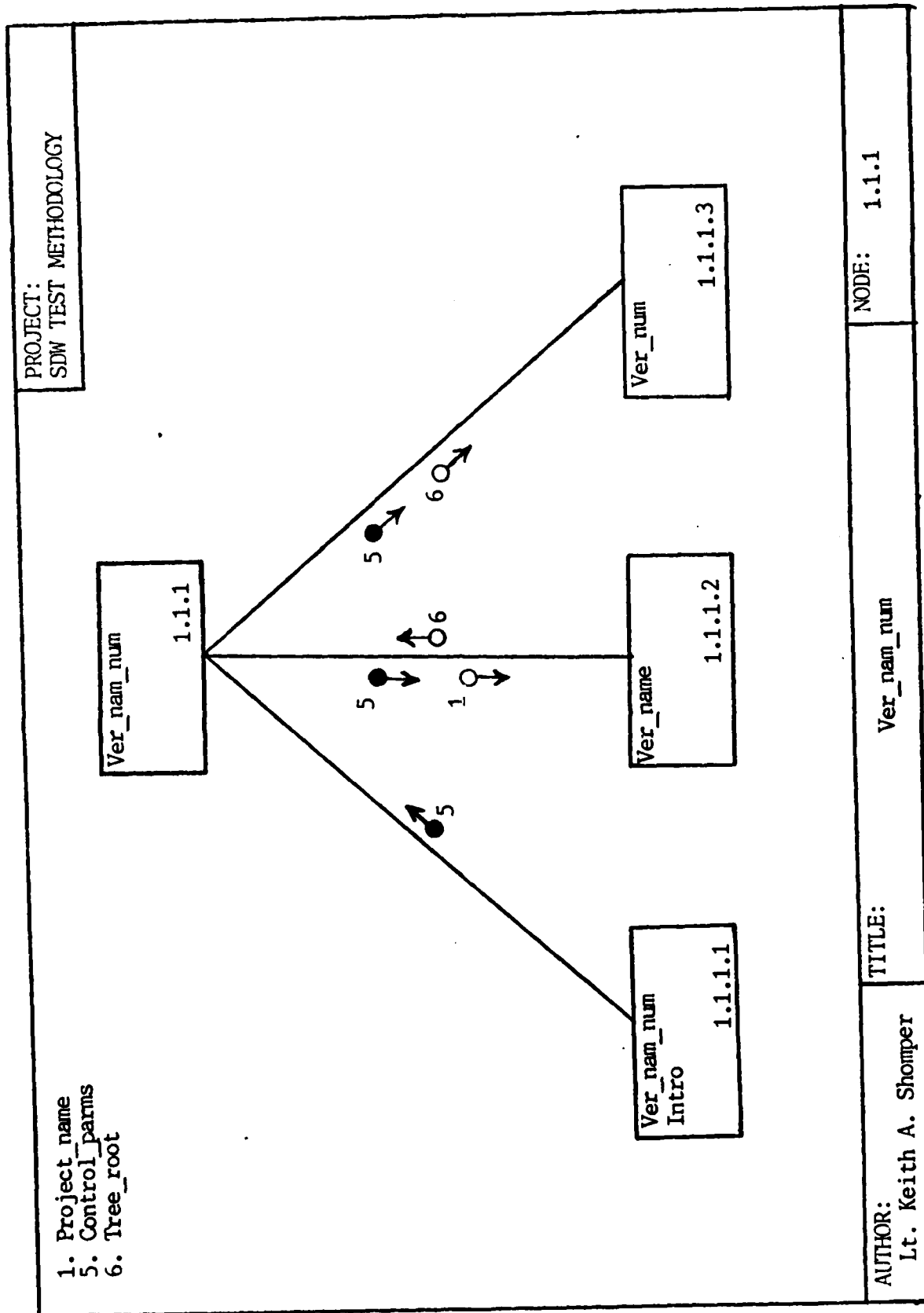
A253

Detailed Design  
Select Command

A253

select module:

using the command word select the appropriate  
module to execute  
END select module:



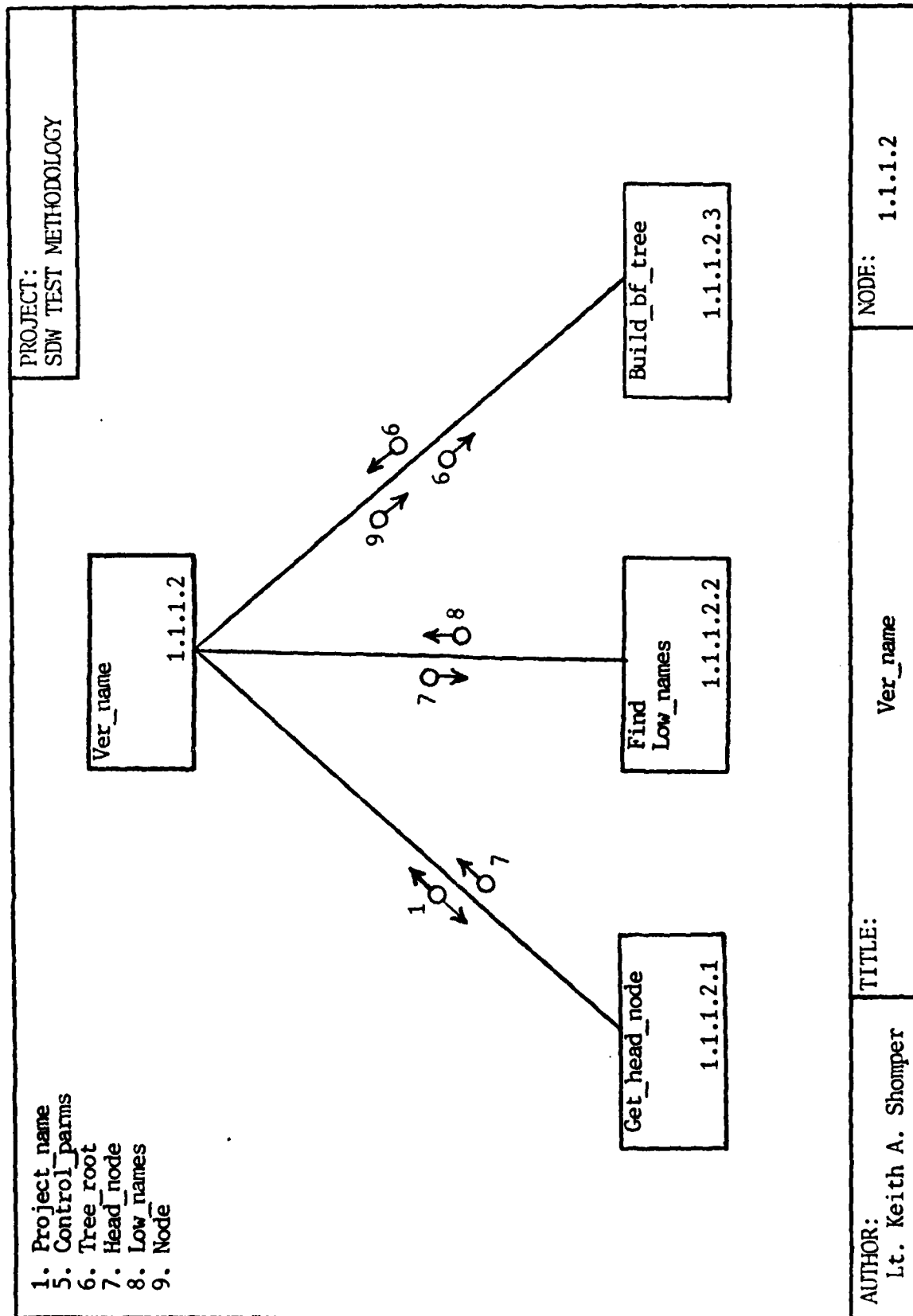
1.1.1

Detailed Design  
Verify Diagram Name and Number

1.1.1

```
ver nam num:
    call ver nam num intro
    call set up
    perform name ver
    IF unrecoverable error is found
        exit
    ENDIF
    perform num ver
End ver nam num:
```





AUTHOR:  
Lt. Keith A. Shomper

TITLE:  
Ver\_name

NODE:  
1.1.1.2

1.1.1.3

Detailed Design  
Perform Name Verification

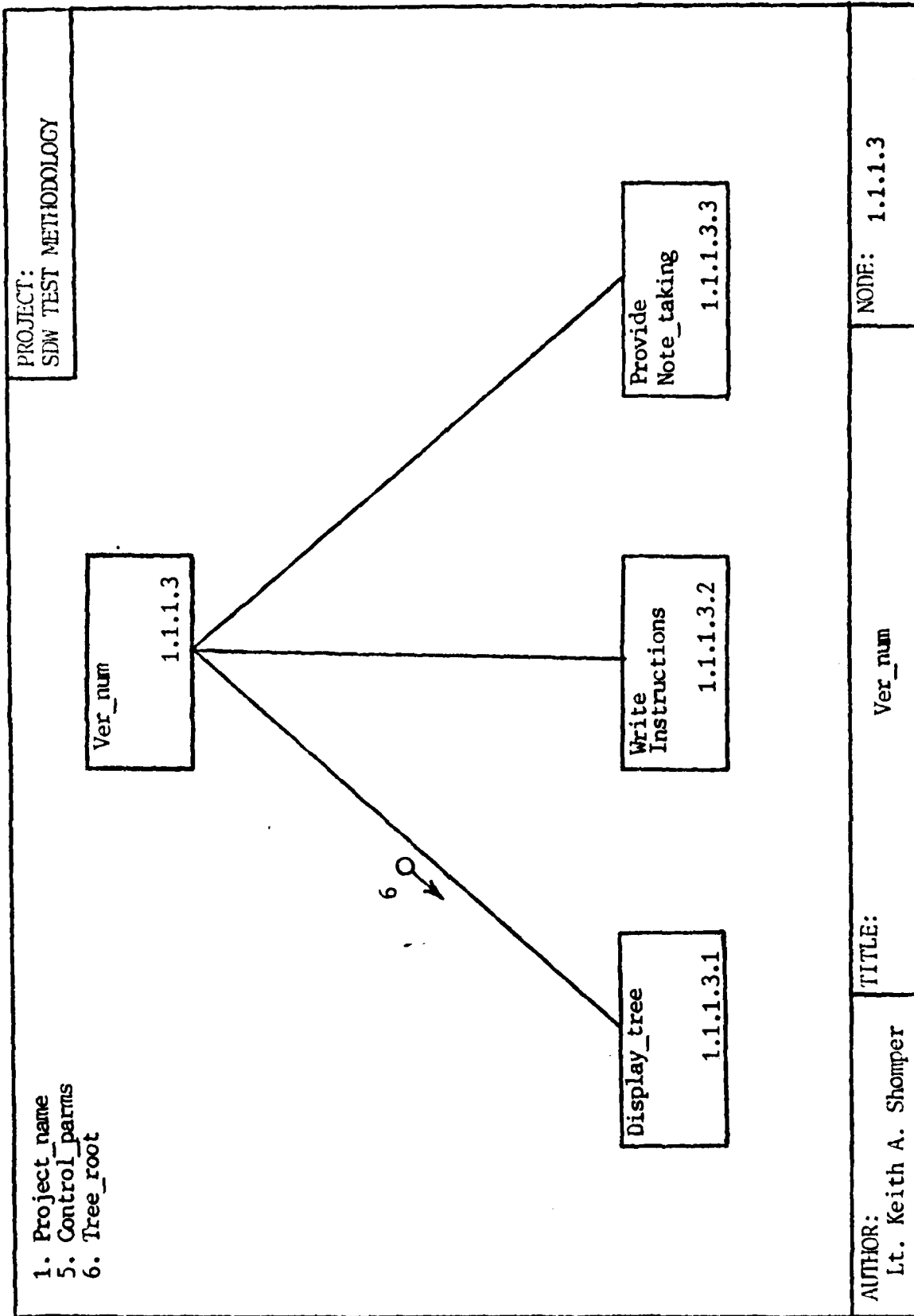
1.1.1.3

ver name:

```
get head node
store head node in process list
DO WHILE name(s) are in process list
    find low names
    add low names to storage list
    IF unrecoverable error is found
        undo
    DO WHILE more low names
        build bf tree
    ENDO
    IF process list is empty
        put storage list in process list
        delete storage list
    ENDIF
```

ENDDO

END ver name:



1.1.1.4

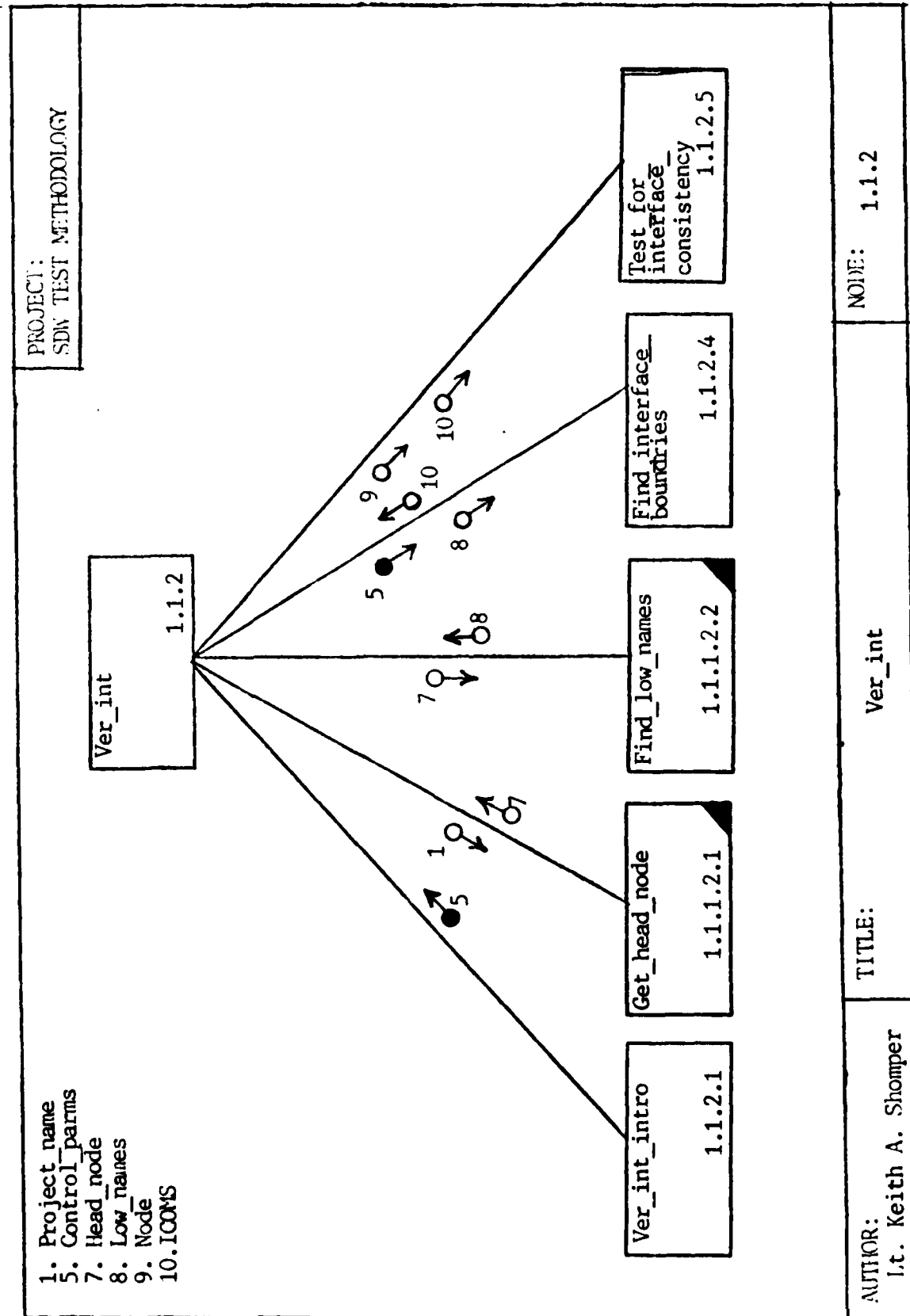
Detailed Design  
Perform Number Verification

1.1.1.4

Ver num

display tree  
write instructions  
provide note taking

END ver num:



1.1.2

Detailed Design  
Verify Interfaces (SADT's)

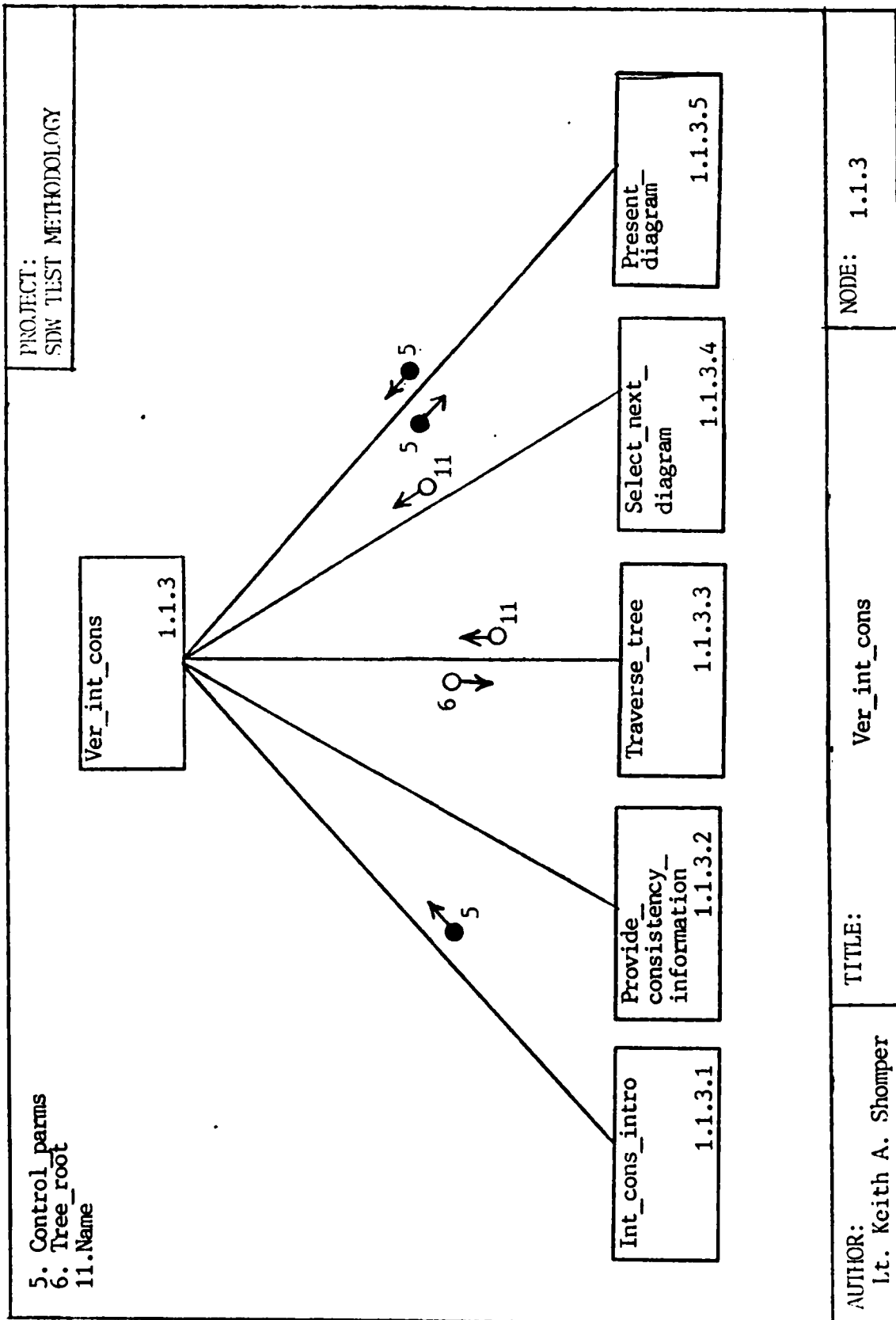
1.1.2

ver int:

```
call ver int intro
call set up
get head node
store head node in process list
DO WHILE name(s) are in process list
    find low names
    add low names to storage list
    DO WHILE more low names
        find interface connections
    ENDO
    find interface boundaries
    test for interface consistency
    IF process list empty
        put storage list in process list
        delete storage list
    ENDIF
```

ENDD

END ver int:



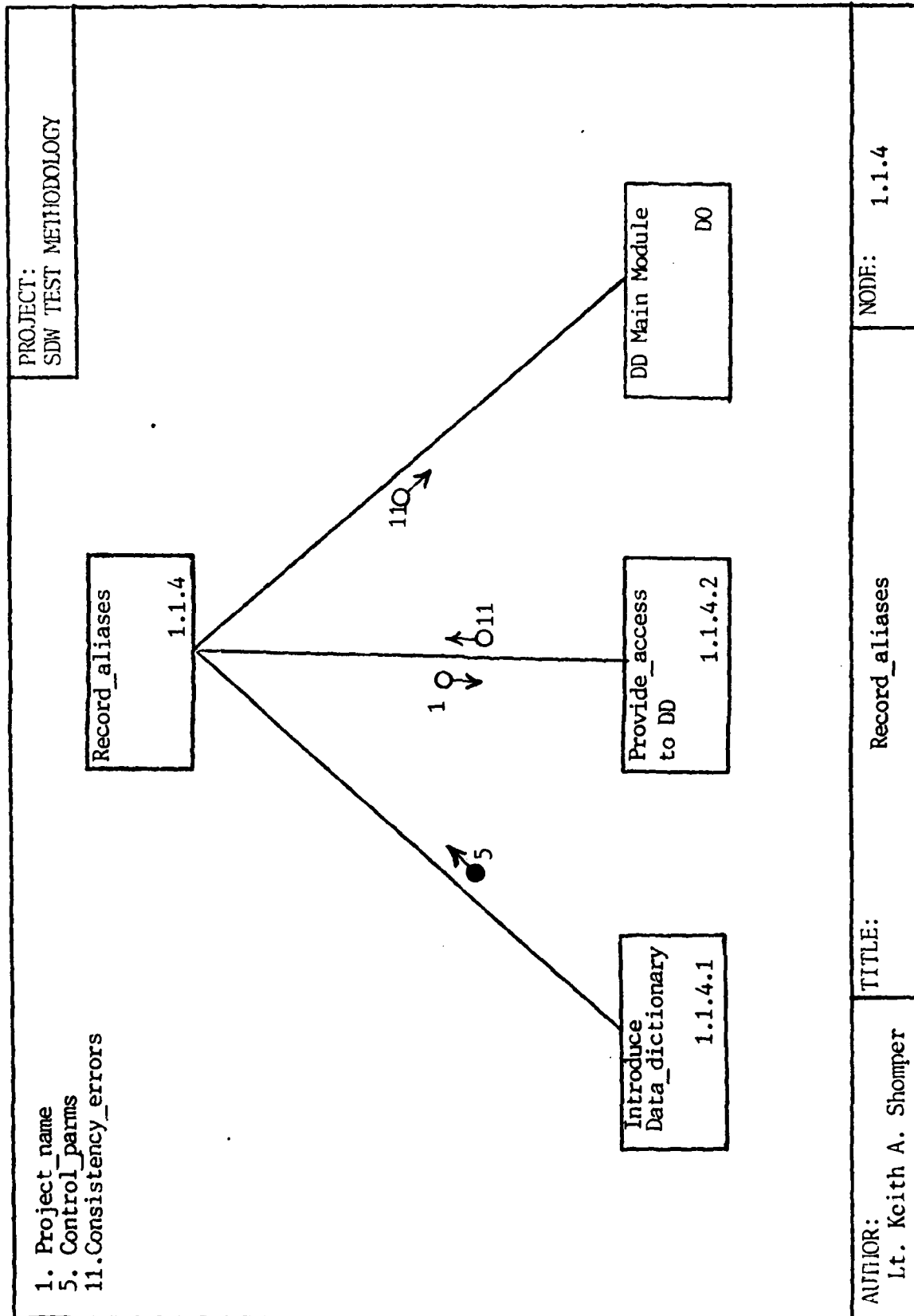
1.1.3

Detailed Design  
Verify Internal Consistency

1.1.3

```
ver int cons:
    int cons intro
    set up
    provide consistency information
    DO WHILE more nodes in tree root or quit is false
        traverse tree
        select next diagram
        present diagram
    ENDDO
END ver int cons:
```





1.1.4

Detailed Design  
Record Aliases

1.1.4

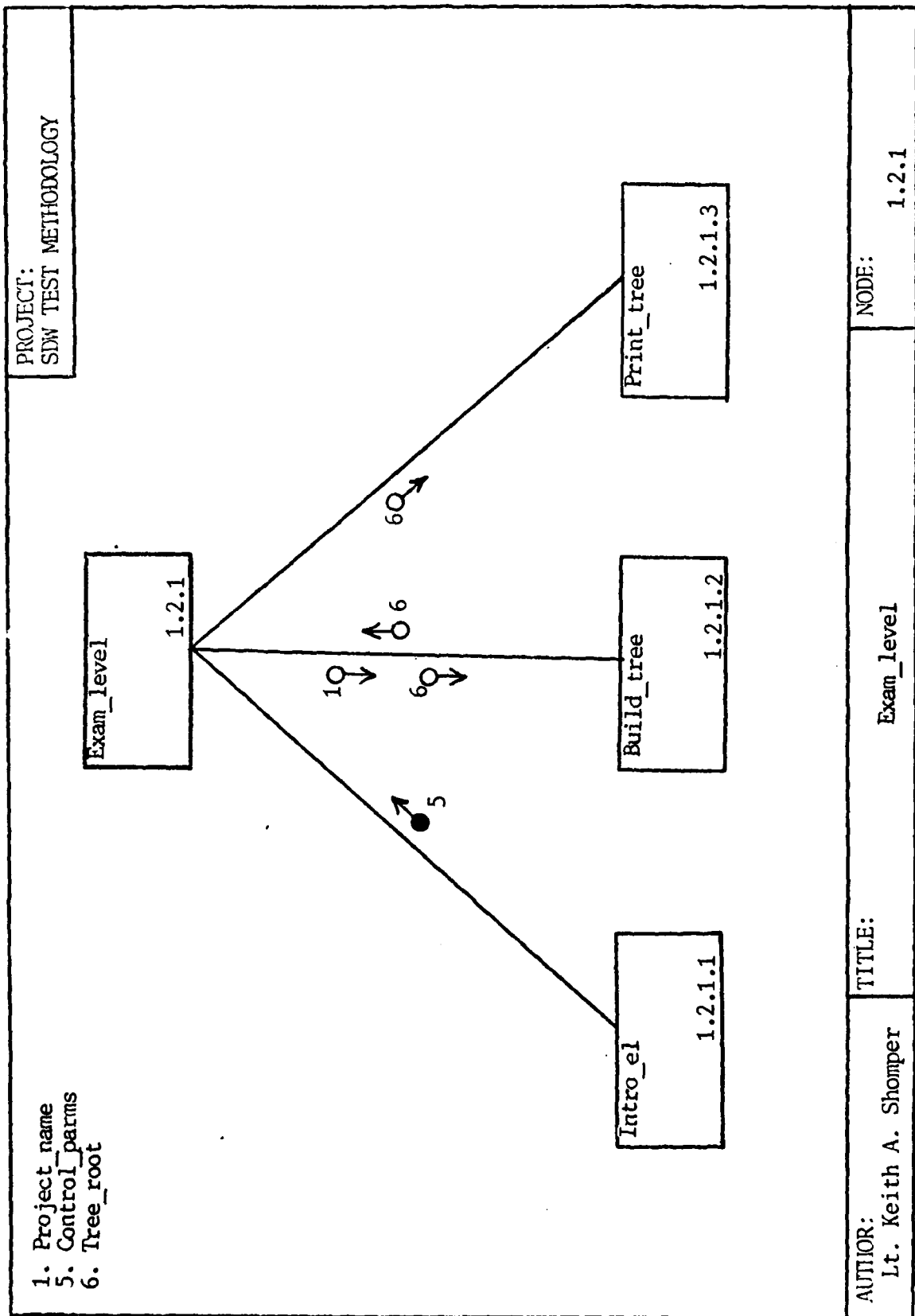
record aliases:

introduce data dictionary  
provide access to data dictionary top end  
call data dictionary

END

record

aliases:

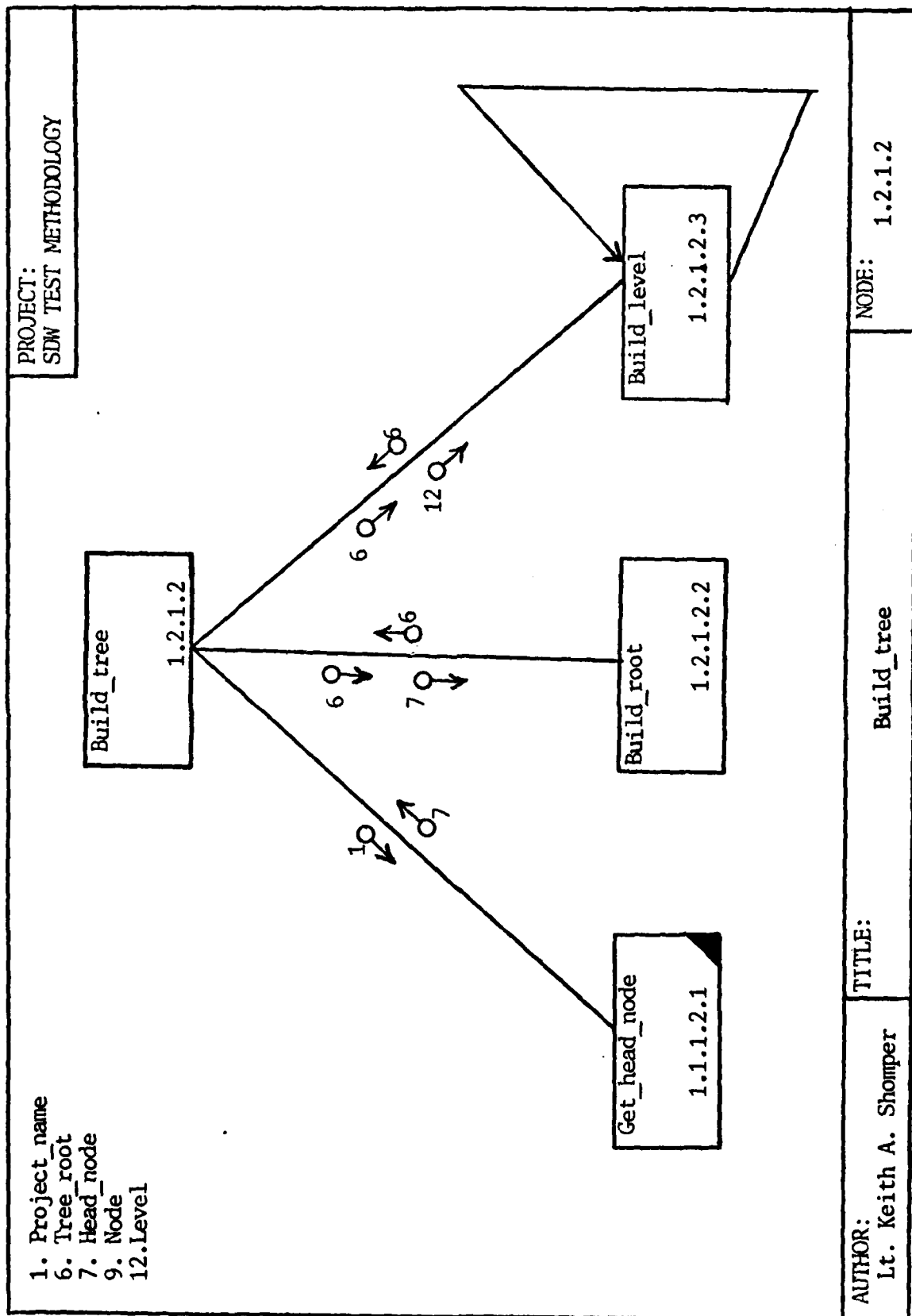


1.2.1

Detailed Design  
Examine Leveling

1.2.1

```
examine leveling:
    introduce exam level
    IF tree root is nil
        build tree
    ENDIF
    print tree
END examine leveling:
```



1.2.1.2

Detailed Design  
Build Tree

1.2.1.2

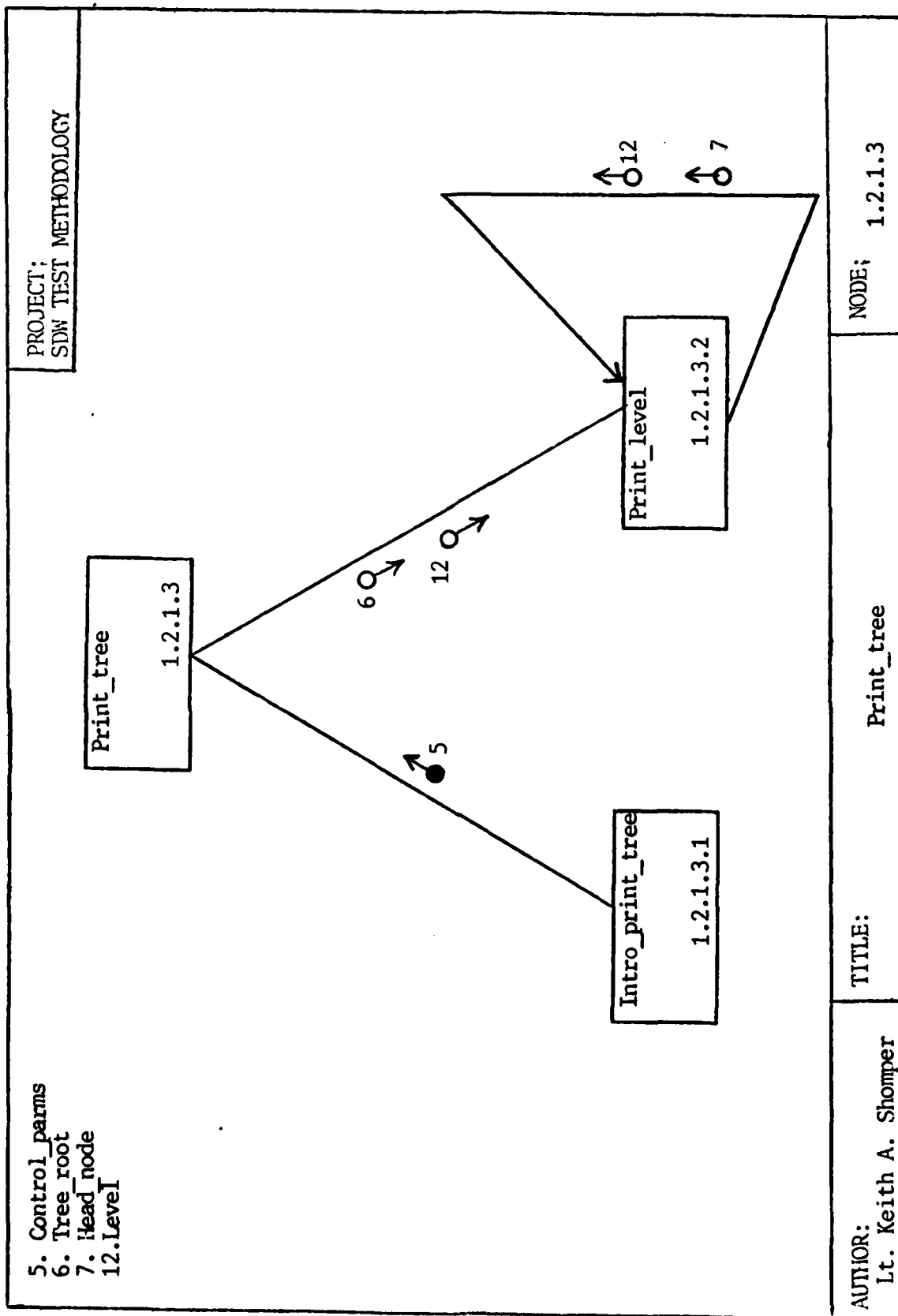
build tree:

    get head node

    build root

    build level

END build tree:



1.2.1.3

Detailed Design  
Print Tree

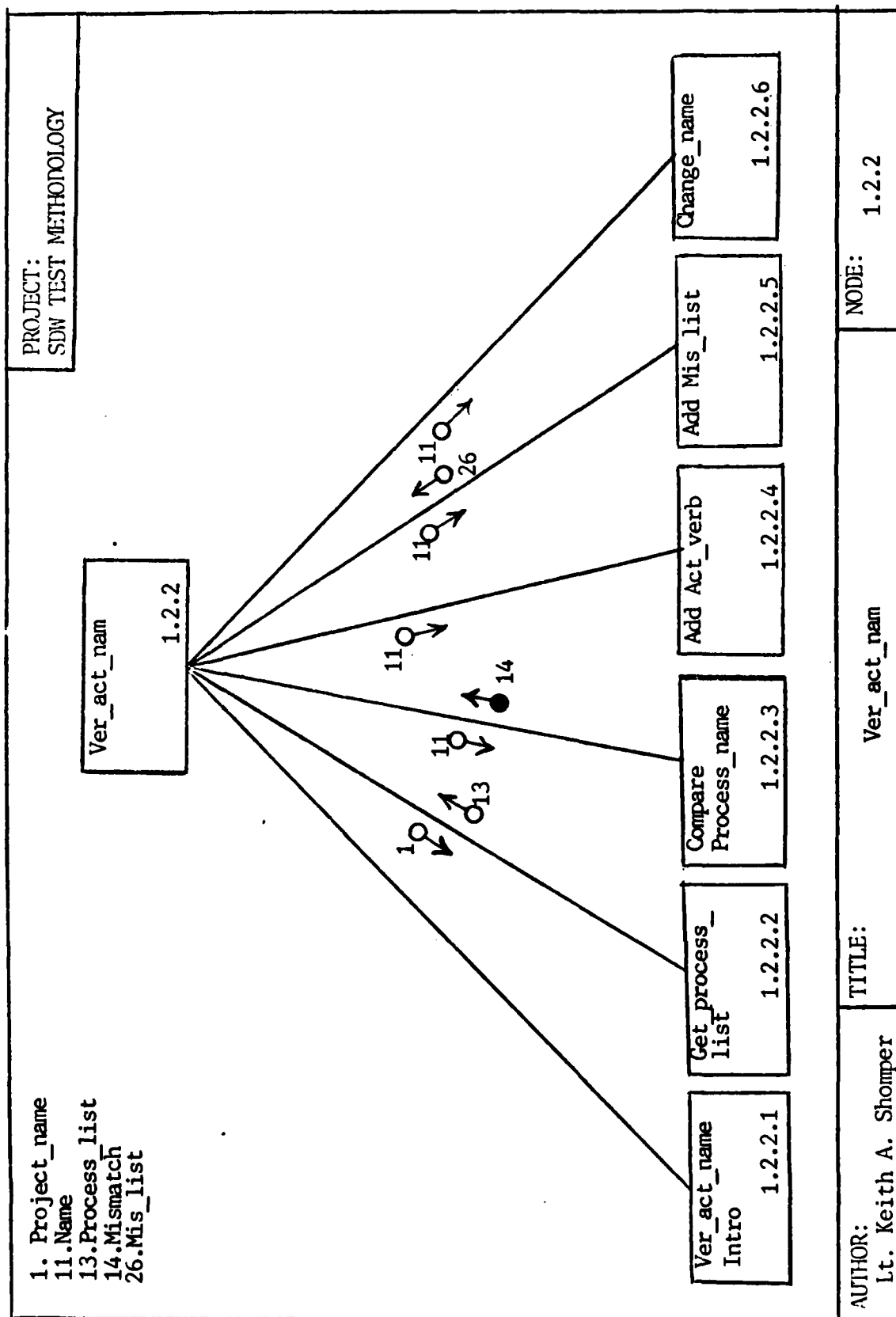
1.2.1.3

print tree:

intro print tree  
set initial level to top level  
top level has one node  
increment level by 1  
call print level

END print tree:



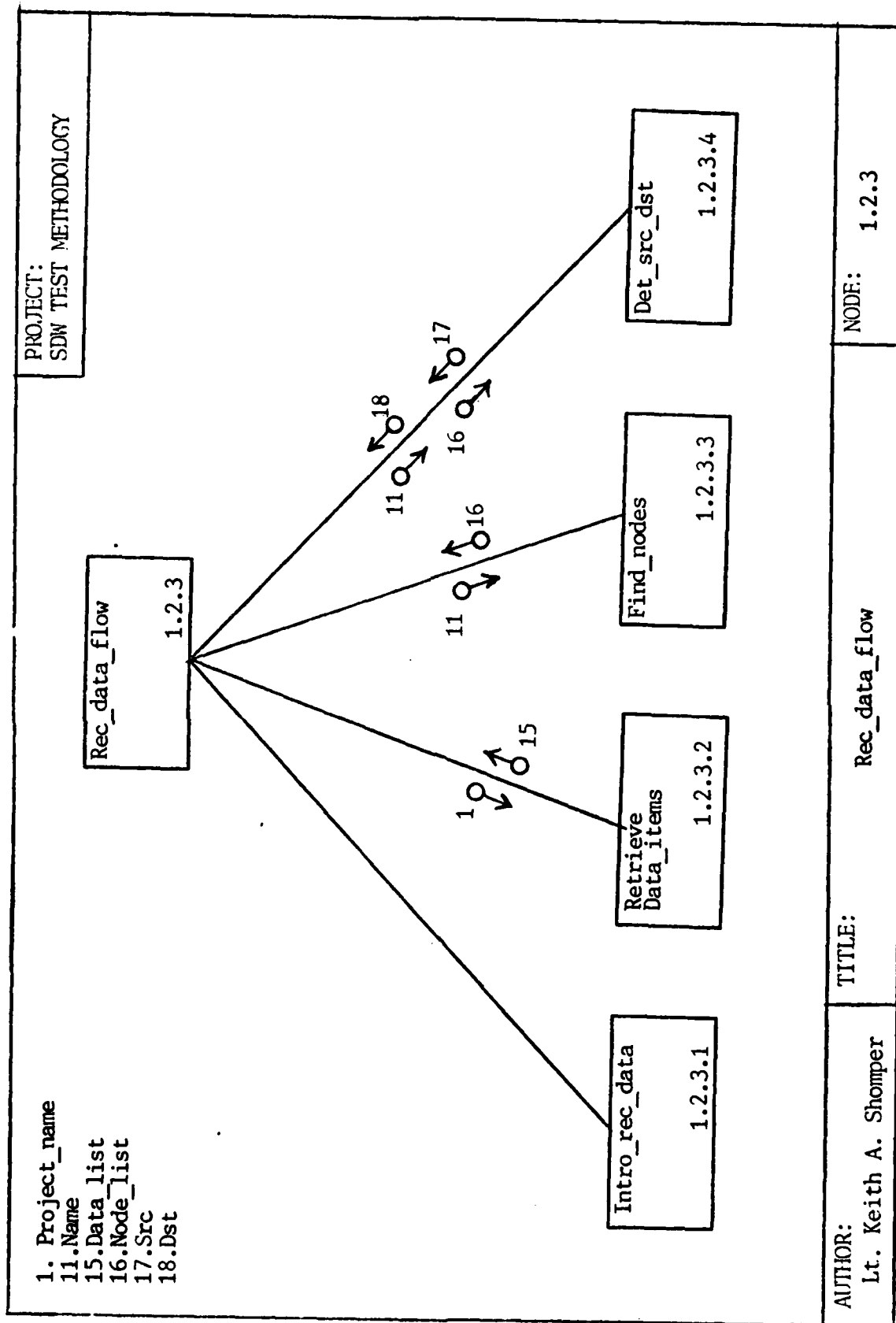


1.2.2

Detailed Design  
Verify Active Process Names

1.2.2

```
ver act nam:
    introduce ver act nam
    retrieve process name list
    DO WHILE names in process name list
        compare process name with active
verb list
    IF process name beginnd with an
active verb
        add active verb to active verb lis
    ELSE
        place process name in mismatched list
    ENDIF
    DO WHILE name in mismatched list
        change process name in database
    ENDO
END ver act nam:
```



1.2.3

Detailed Design  
Record Data Flow

1.2.3

rec data flow:

introduce rec data flow

retrieve data items

DO WHILE more data items

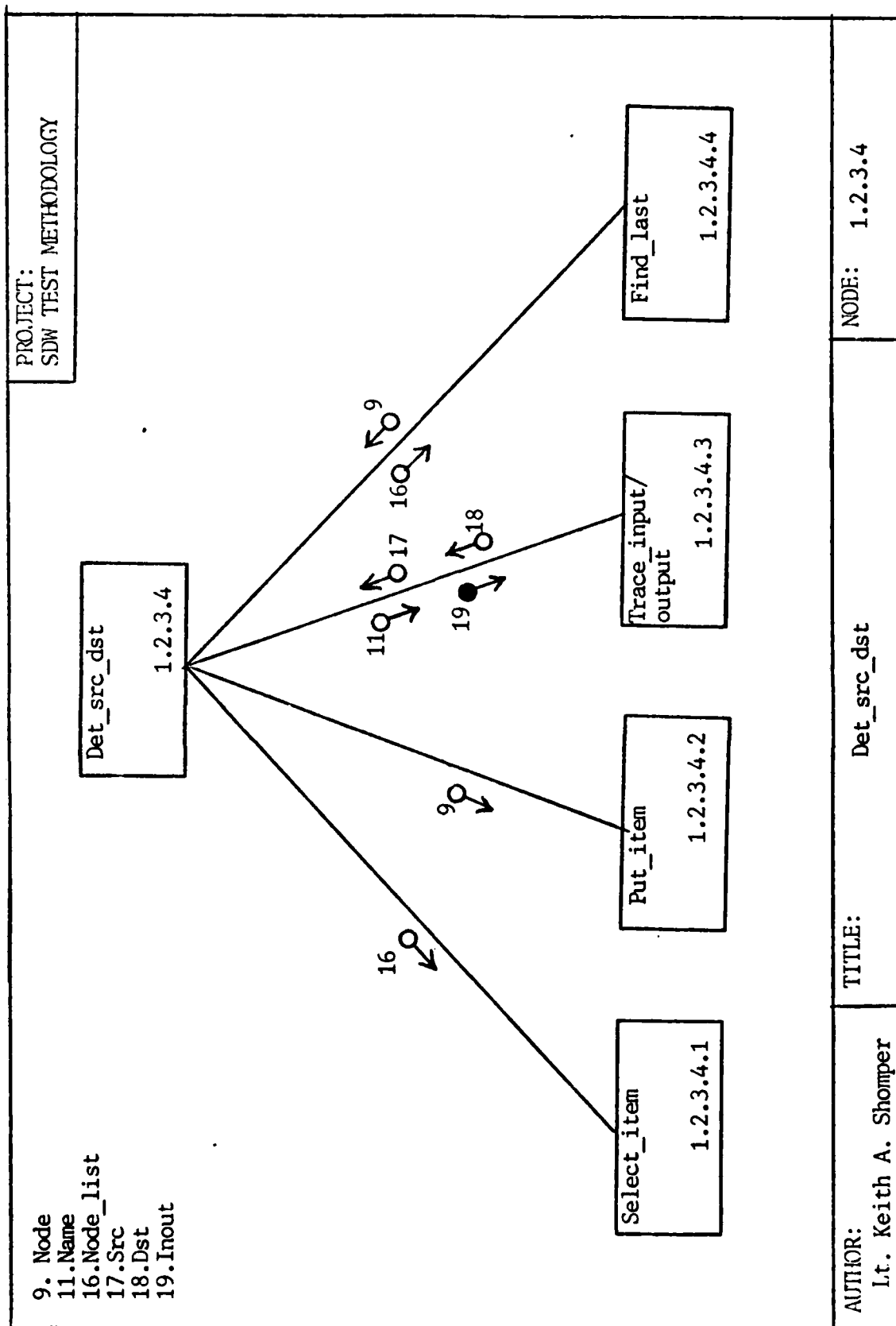
find nodes associated with this

data item

determine sources and destinations

ENDDO

END rec data flow:

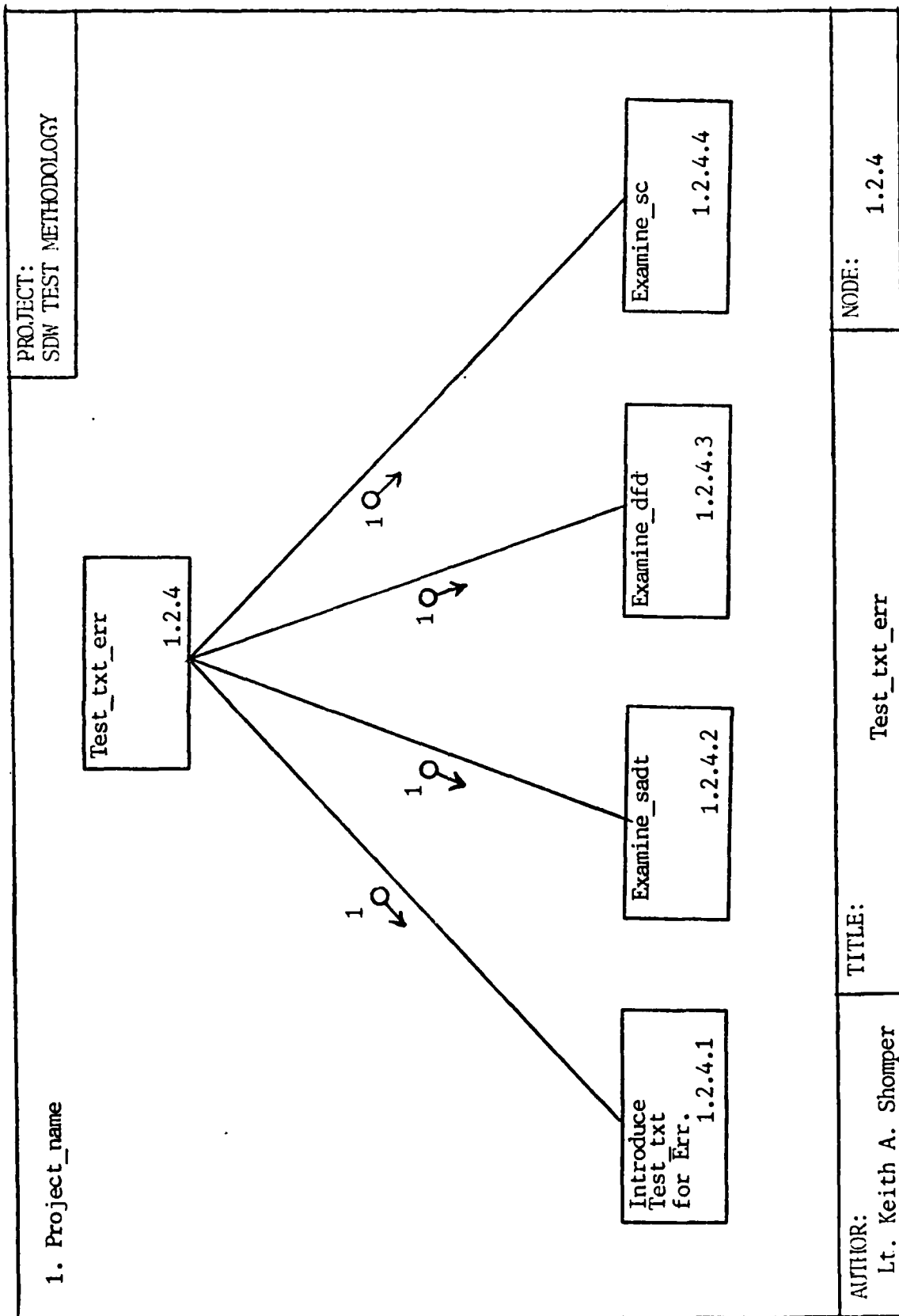


1.2.3.4

Detailed Design  
Determine Source and Destination

1.2.3.4

```
det src dst:
  select data item
  IF data item is input
    DO WHILE more data items
      put input item in path tree
      select data item
    ENDO
    trace input item through path tree
  ELSE
    DO WHILE more data items
      put output item in path tree
      select data item
    ENDO
    trace output item through path tree
  ELSEIF data item is control or mechanism
    find last output path
    trace output item through path tree
  ENDIF
END det src dst:
```



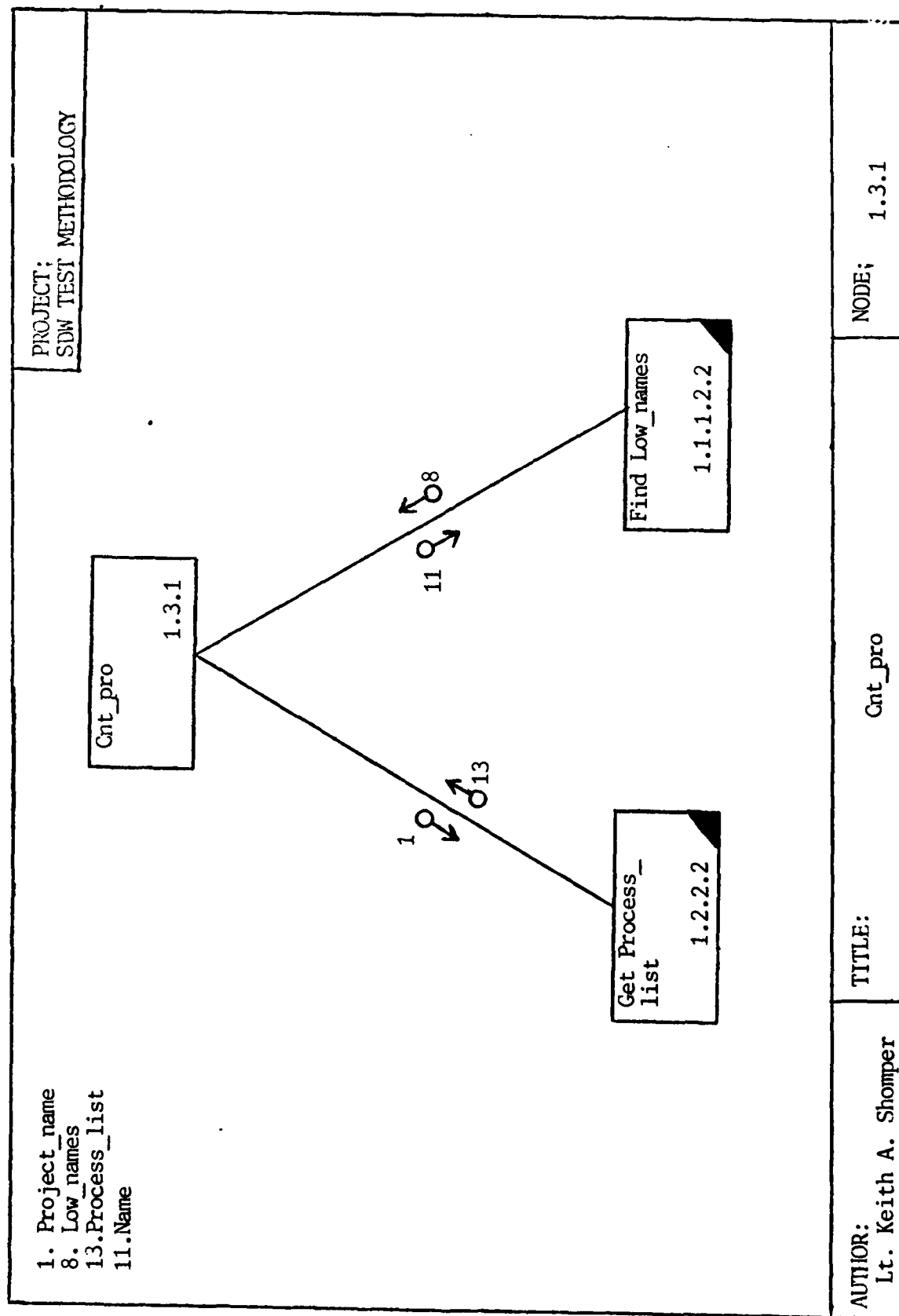
1.2.4

Detailed Design  
Test For Textual Errors

1.2.4

```
test txt err:
  introduce test txt err
  IF reqmnts phase
    IF sadt
      present sadt info file
      examine sadt
      present diagram
    ELSE
      present dfd info file
      examine dfd
      present diagram
    ENDIF
  ELSE
    present sc info file
    examine structure charts
    present diagram
  ENDIF
end test txt err:
```





1.3.1

Detailed Design  
Count Processes

1.3.1

cnt pro:

get process list

DO WHILE more processes

find low name

DO WHILE more low names

count:=count+1

ENDO

IF count>6

report error

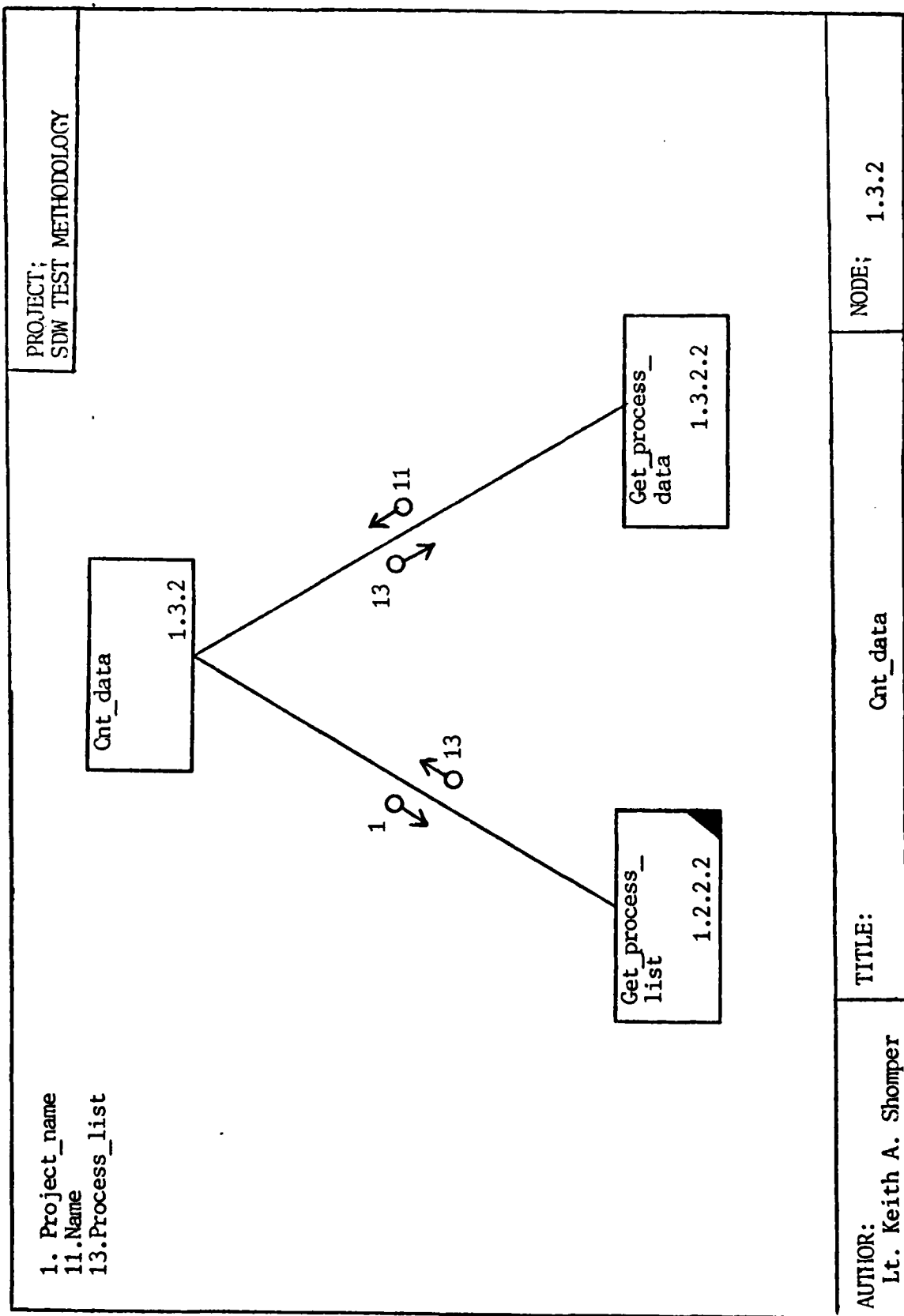
record error

count=0

ENDIF

ENDO

END cnt pro:



1.3.2

Detailed Design  
Count Data Items

1.3.2

cnt data:

retrieve process list  
DO WHILE more process items  
    get process data  
    DO WHILE more data items

    END  
    IF count>6

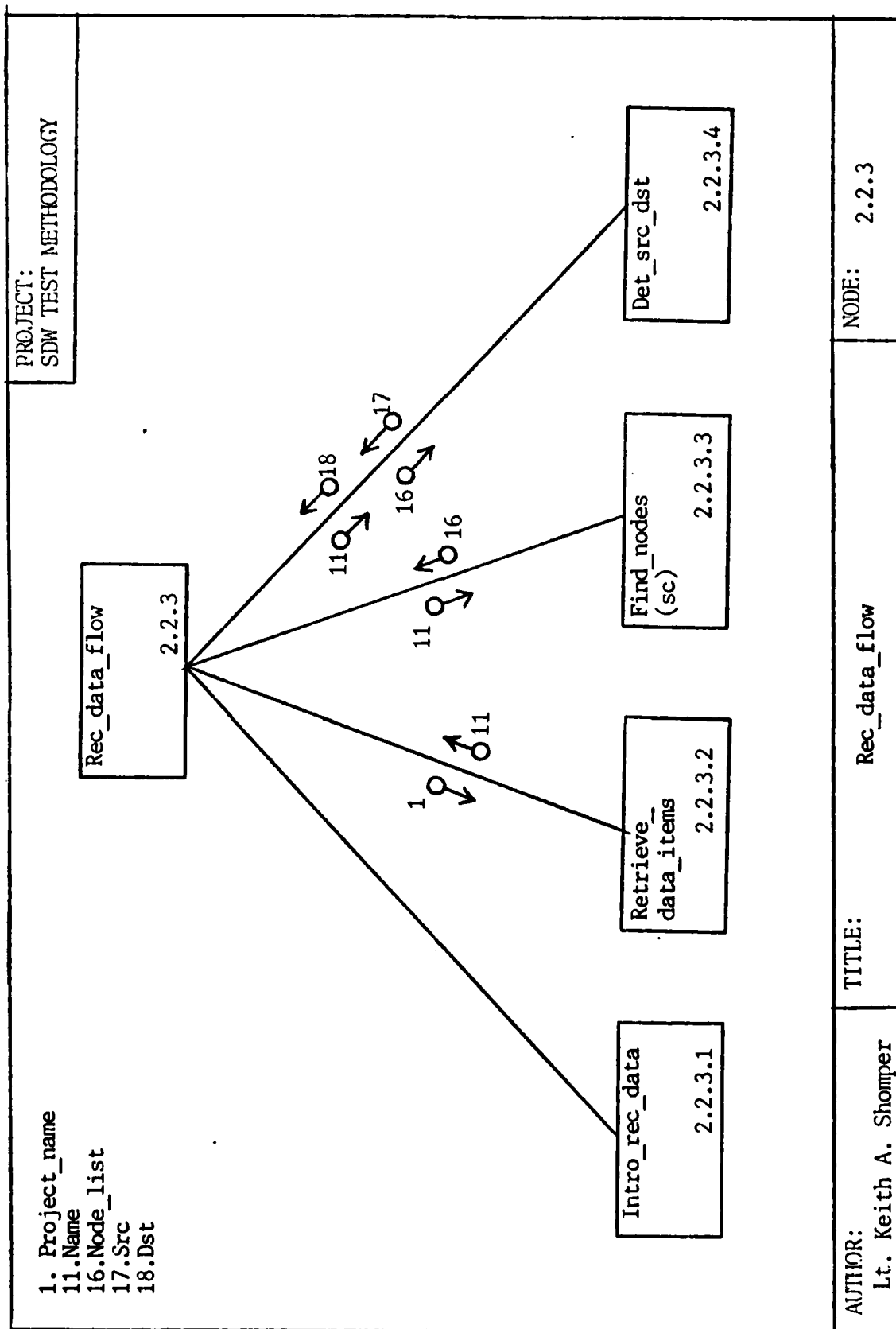
    ENDIF  
    count=0

END

END cnt data:

count:=count+1

report error  
record error



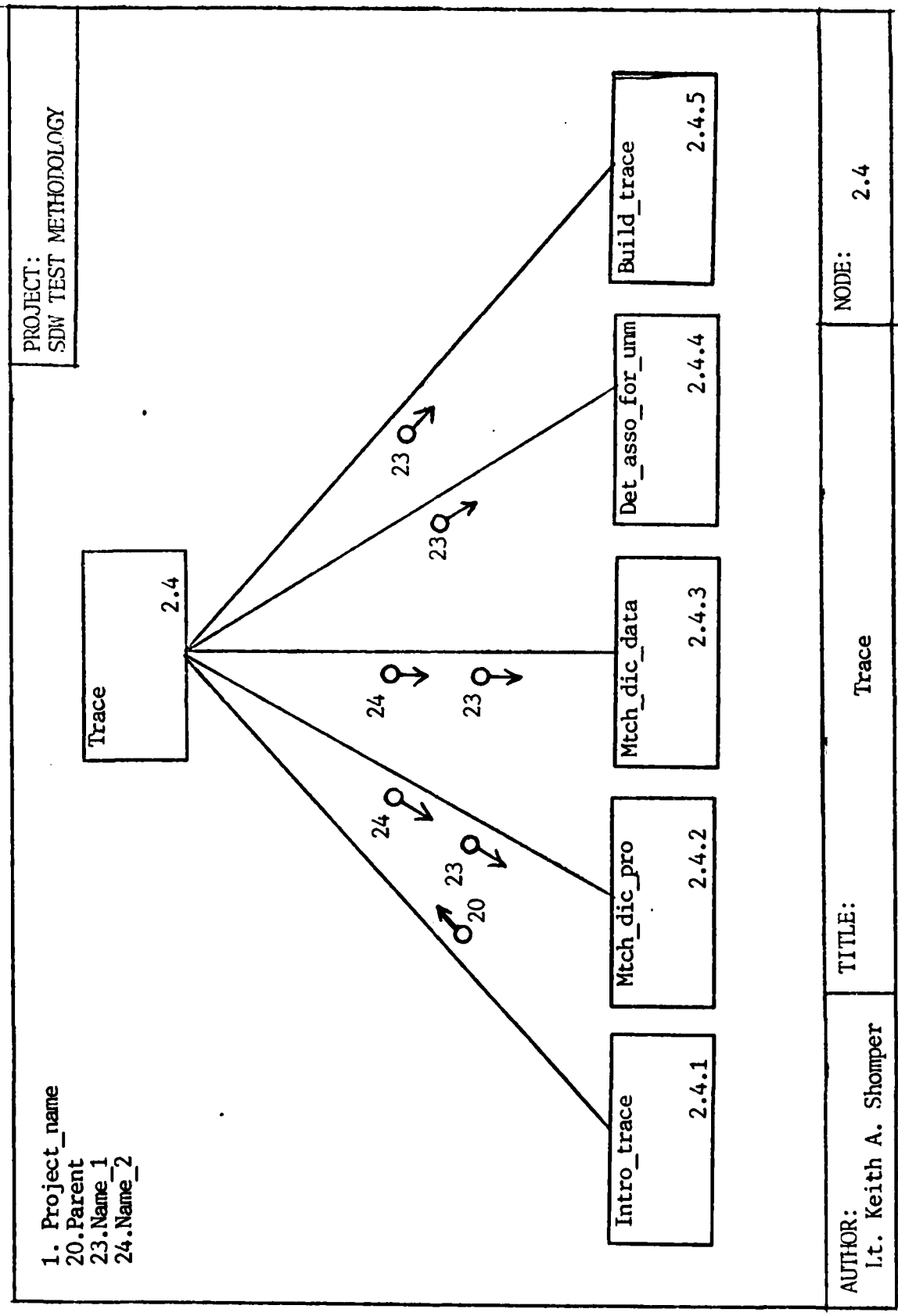
2.2.3

Detailed Design  
Record Data Flow

2.2.3

rec data flow:

```
    intro rec data
    retrieve data items
    DO WHILE more data items
        find associated processes (passed
to/from)
        determine each items source and
destination, by structure chart scheme
    ENDO
END rec data flow:
```



trace:

intro to trace tests

match dictionary names for processes in parent  
current models

and

match dictionary names for data in parent and  
models

current

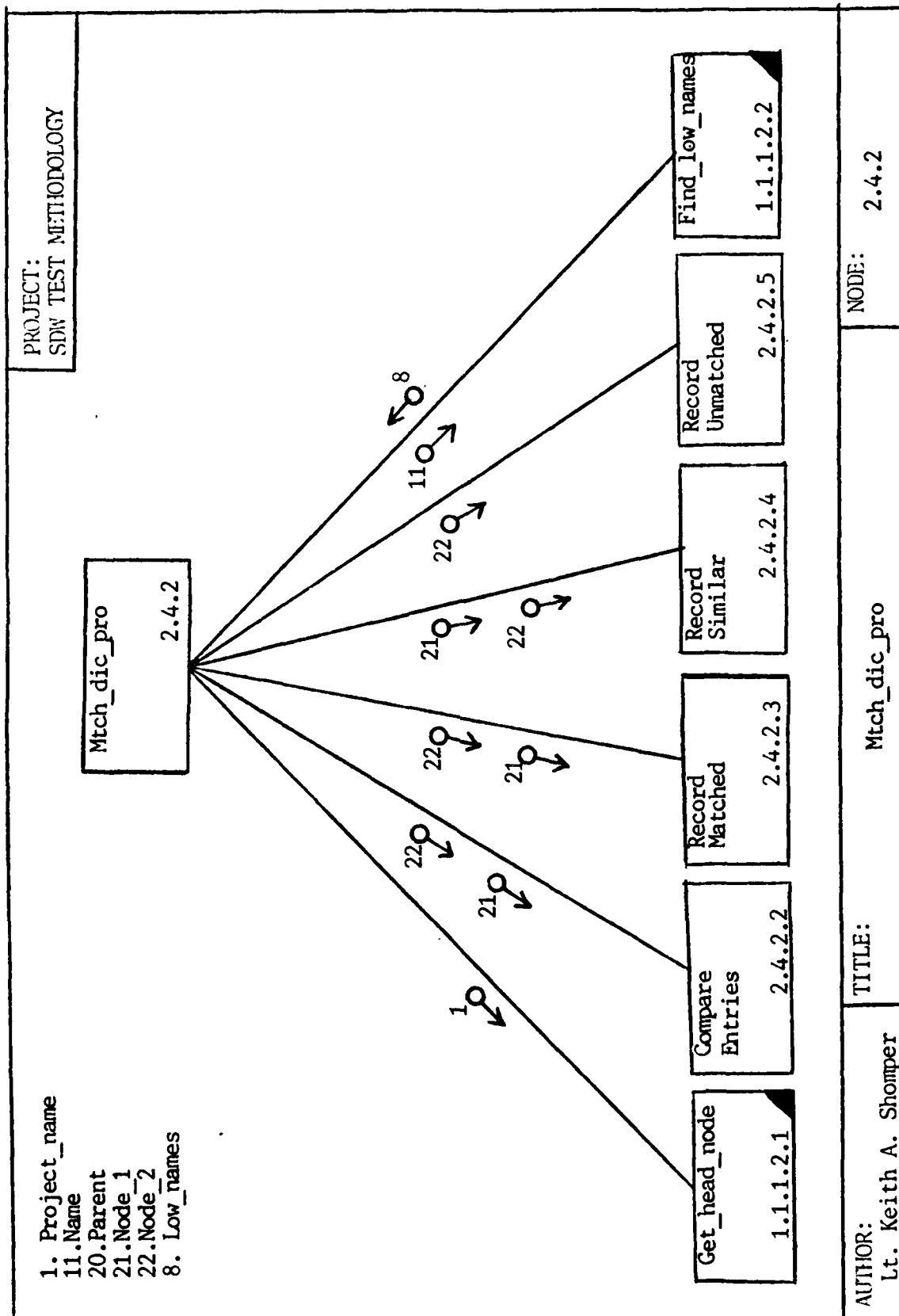
for all unmatched entries, find an association  
with parent model

for each

build traceability database

END trace:



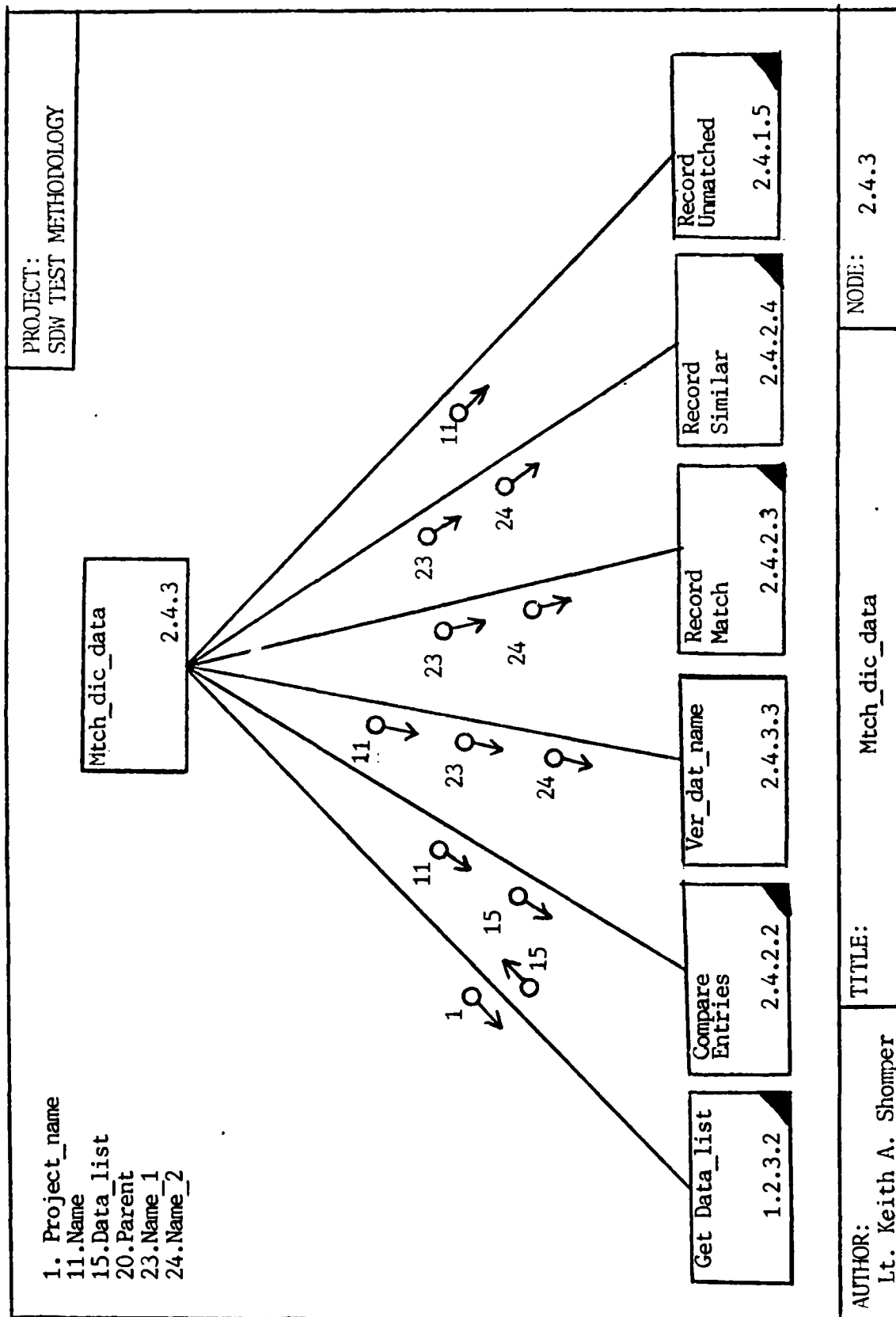


2.4.1

Detailed Design  
Match Dictionary Entries (Processes)

2.4.1

```
mtch dic pro
  intro to mtch dic
  get head node of process module
  get head node of current model
  DO WHILE still more names to compare
    compare model names wi
    IF model names match
      record matches
    ELSEIF they almost match
      record them as similarities
    ELSE
      record them as unmatched process names
    ENDIF
    when the current list of names is
      exhausted get the next
      lower list (find low names)
  ENDO
  make one last check among the singles for matches
END mtch dic pro:
```



AUTHOR:  
Lt. Keith A. Shomper

TITLE:  
Mtch\_dic\_data

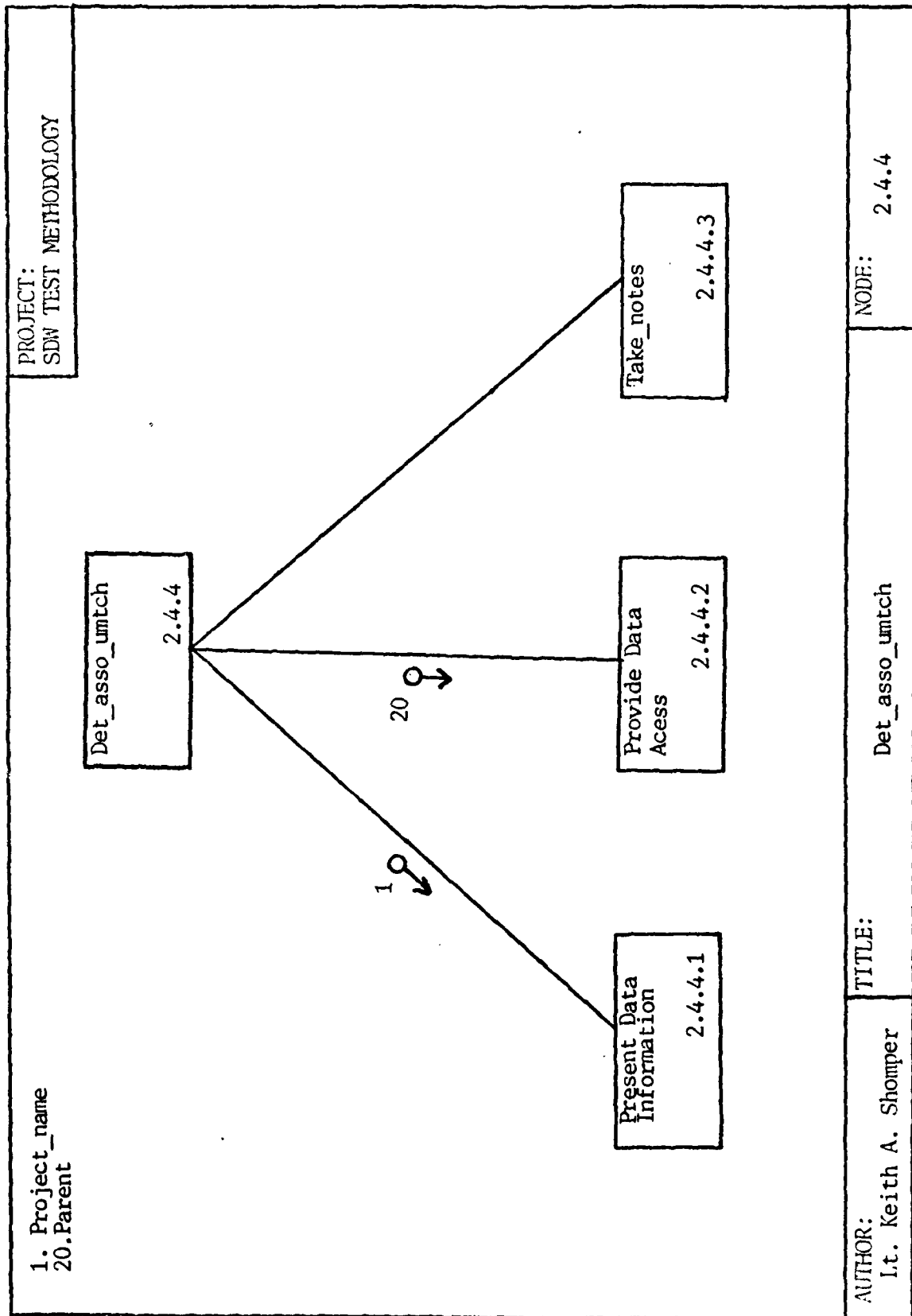
NODE:  
2.4.3

2.4.2

Detailed Design  
Mtch Dic Data

2.4.2

```
mtch dic data
    intro mtch dic data
    get data list for parent model
    get data list for current model
    DO WHILE there are more names in the current
model list
        compare data name with parent model
list
        verify name
        IF data name matches between models
            record match
        ELSEIF they are similar
            record similar
        ELSE
            record it as an unmatched entry
        ENDIF
    ENDO
END mtch dic data:
```



2.4.3

### Detailed Design

2.4.3

#### Determine Associations For Unmatched Entries

det asso umtch:

DO WHILE there are more unmatched entries

present data information

provide data access to parent

project

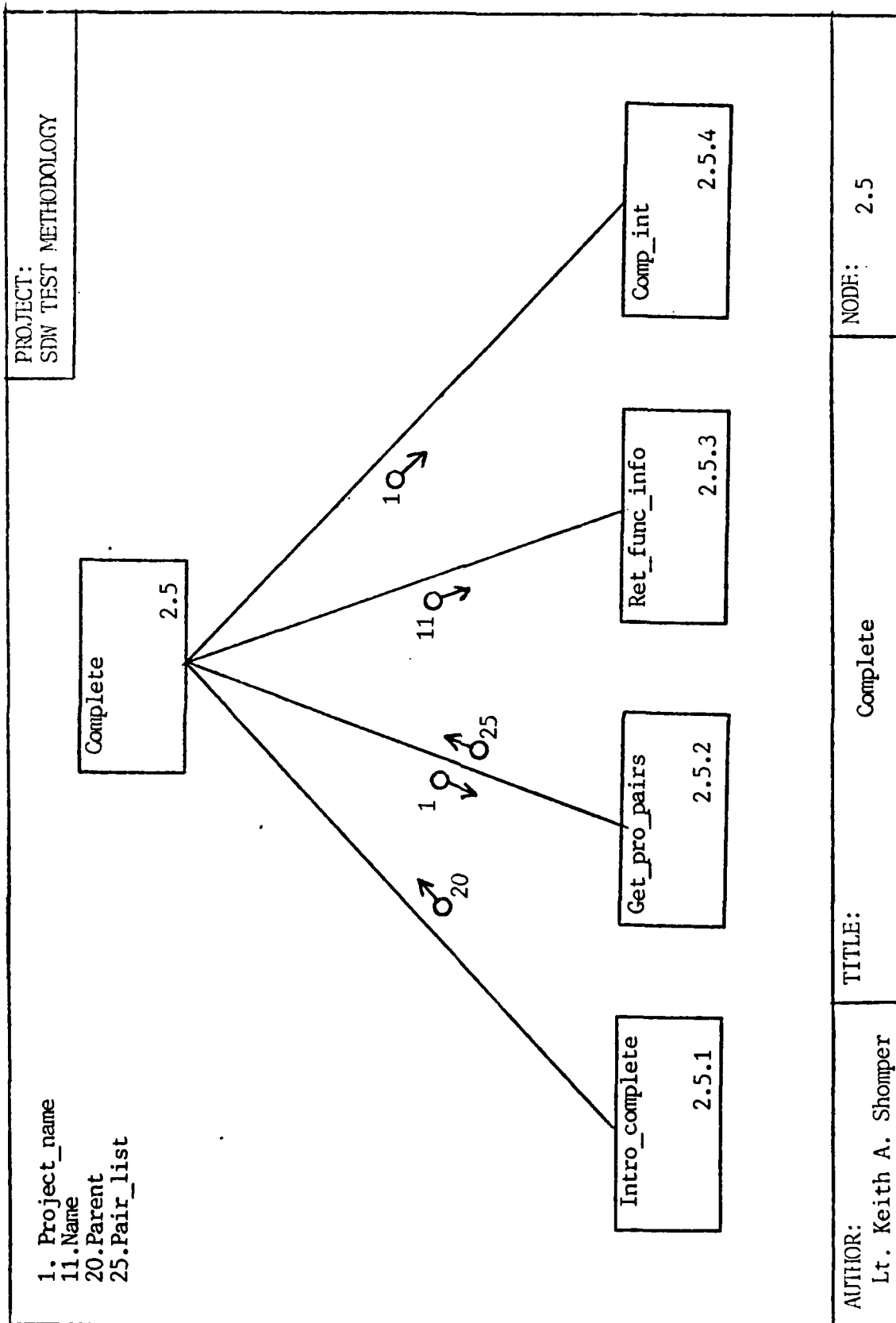
allow user to take my notes or

comments necessary on

this data item

ENDO

END det asso umtch:



2.5

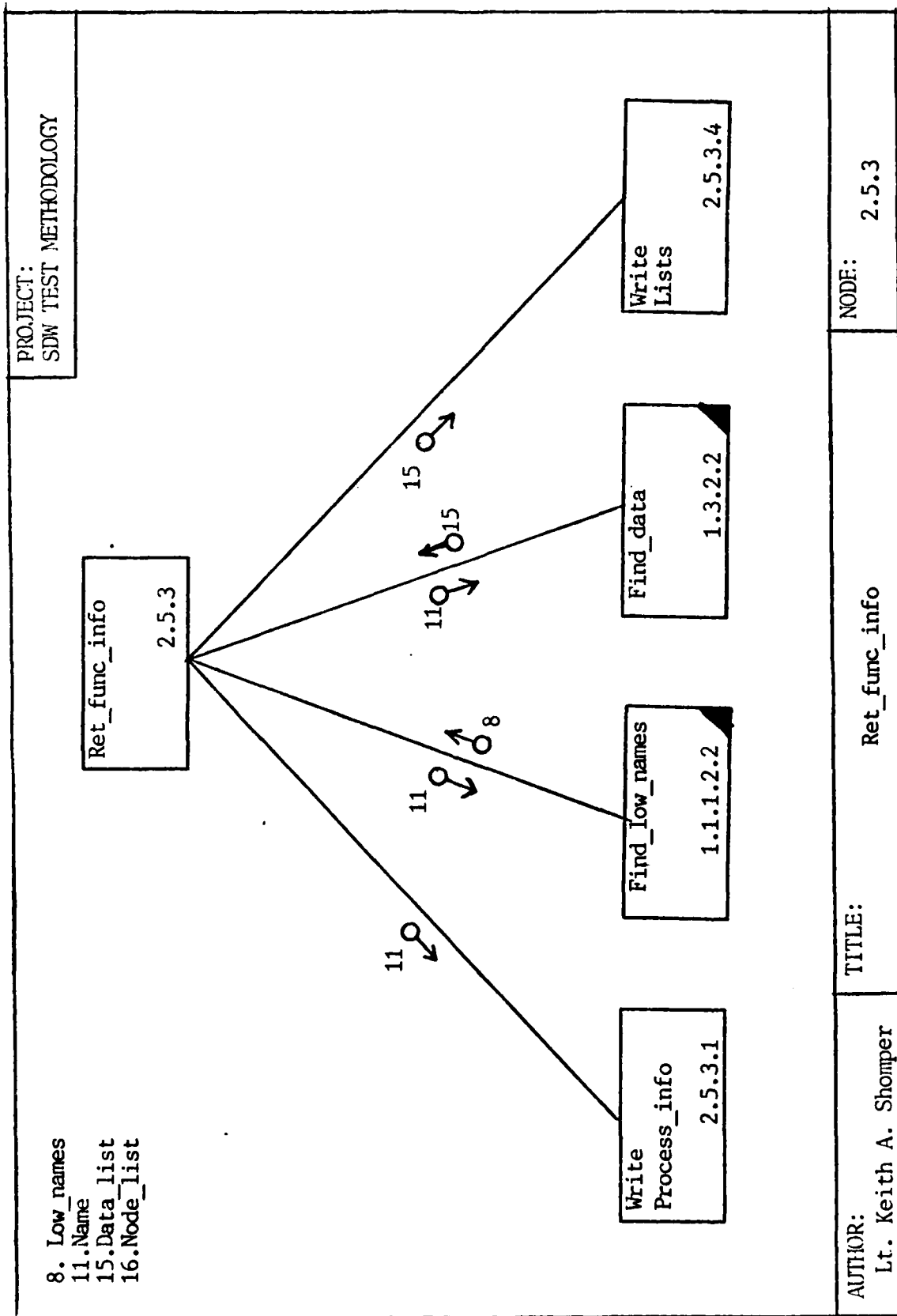
Detailed Design  
Completeness Tests

2.5

complete:

```
intro to complete
get process pair list from that file created in
the traceability tests
DO WHILE there are more pairs in the list
functional info      retrieve parent model process
functional info      retrieve current model process
functional info      compare functional intent
                        ENDO
END complete:
```





2.5.3

Detailed Design  
Retrieve Process Functional Information

2.5.3

ret func info:

write process info to screen

find that processes low names

write these lists to the screen also

END ret func info:

1. Project_name		PROJECT; SDW TEST METHODOLOGY	
<div style="display: flex; justify-content: space-around; align-items: center;"> <div style="border: 1px solid black; padding: 5px; text-align: center;">Comp_int 2.5.4</div> <div style="display: flex; flex-direction: column; align-items: center;"> <div style="margin-bottom: 20px;">1</div> <div style="font-size: 2em;">↘</div> </div> <div style="border: 1px solid black; padding: 5px; text-align: center;">Write Message 2.5.4.1</div> </div> <div style="display: flex; justify-content: space-around; align-items: center; margin-top: 20px;"> <div style="border: 1px solid black; padding: 5px; text-align: center;">Take_notes 2.5.4.2</div> </div>			
AUTHOR: Lt. Keith A. Shomper		TITLE: Comp_int	
		NODE: 2.5.4	

2.5.4

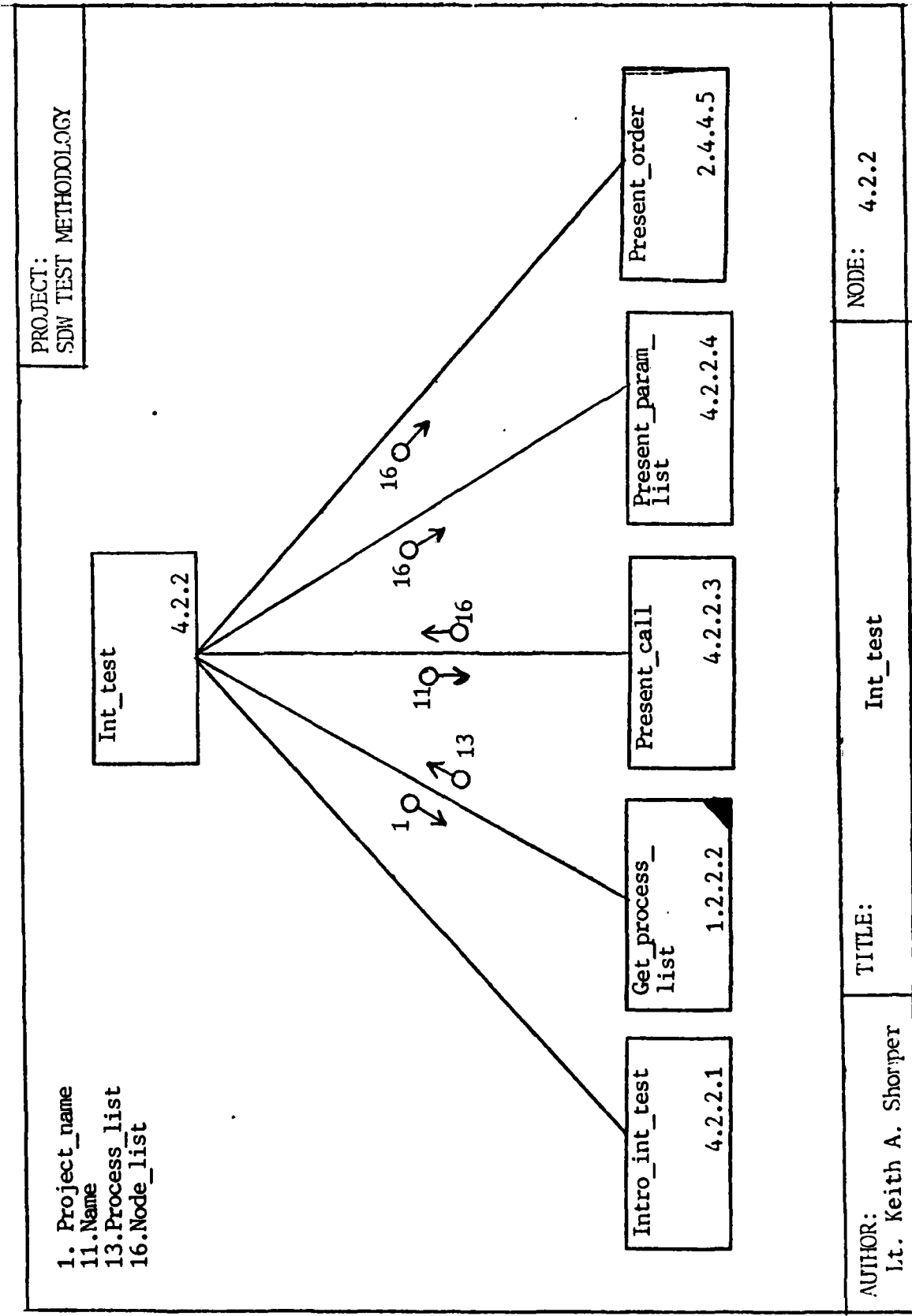
Detailed Design  
Compare Functional Intent

2.5.4

comp int:

write message about functional testing and prompt  
allow the user the capability to take notes

END comp int:



4.2.2

Detailed Design  
Provide Interface Testing Management

4.2.2

int test:

intro int test  
get process list

DO While there are more processes

present the associated procedure or

functions

parameter list

present the defined order of

parameters

ENDO:

END int test:

Appendix D  
Data Dictionary

## Appendix D

### Data Dictionary

#### Introduction

The data dictionary is written to support the Detailed Design in this investigation. This appendix contains cross reference and descriptive information concerning each of the Design's process and data items. Although, the Data Dictionary contains much of the same information as the Detailed Design, it is useful as an alternative for design documentation because it also gives descriptive information for each process and data item. Whereas the Detailed Design is to show hierarchy and control flow, the Data Dictionary's main purpose is to provide the descriptive information and parameter specifications. The data items and process items are listed alphabetically herein.



Data Item

NAME: command  
ALIASES: n/a  
DESCRIPTION: A string variable which contains the current  
                  command.  
SOURCES: A23, A24, A261  
DESTINATIONS: A24, A26, A261, A262  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, length 6  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: command buffer  
ALIASES: n/a  
DESCRIPTION: A concatenated encoding of the command words  
                  given thus far. This variable is used for  
                  retracing a user's position back through the  
                  menu.  
SOURCES: A2, A261  
DESTINATIONS: A22, A25, A26, A261  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, length 20  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: consistency errors  
ALIASES: n/a  
DESCRIPTION: String messages which report on the condition  
of test results.  
SOURCES: 1.1.4.2  
DESTINATIONS: D0  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 80  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: control parms  
ALIASES: n/a  
DESCRIPTION: A character string of boolean values which  
controls the attribute settings of particular  
component tests.  
SOURCES: 1.1.1.1, 1.1.2.1, 1.1.3.1, 1.1.3.5., 1.1.4.1,  
1.2.1.1, 1.2.1.3.1  
DESTINATIONS: 1.1.1.2, 1.1.1.3, 1.1.2.4, 1.1.3.5  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string of boolean, lenght 20  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: data list  
ALIASES: n/a  
DESCRIPTION: A list of data items, usually those associated  
with a single process.  
SOURCES: 1.2.3.2, 1.3.2.2  
DESTINATIONS: 2.4.2.2, 2.5.3.4  
COMPOSITION: array of names  
PART OF: n/a  
DATA CHARACTERISTIC: array of character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: dst  
ALIASES: n/a  
DESCRIPTION: A process name variable which denotes the  
destination of a data item.  
SOURCES: 1.2.3.4, 1.2.3.4.3, 2.2.3.4  
DESTINATIONS: none  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: head node

ALIASES: n/a

DESCRIPTION: A single record node containing fields for an activity name and number. This variable is designated 'head node' because it often contains the parent node of a number of children.

SOURCES: 1.1.1.2.1, 1.2.1.3.2

DESTINATIONS: 1.1.1.2.2, 1.2.1.2.2, 1.2.1.3.2

COMPOSITION: name and number fields

PART OF: n/a

DATA CHARACTERISTIC: record variable

DATE ENTERED: 12 November 1984

ENTERED BY: Lt Keith A. Shomper

VERSION: 1.0

Data Item

NAME: ICOMS

ALIASES: n/a

DESCRIPTION: An array of data names and their associated characteristics (input, output, control, or mechanism).

SOURCES: 1.1.2.4

DESTINATIONS: 1.1.2.5

COMPOSITION: array of names

PART OF: n/a

DATA CHARACTERISTIC: array of character string, length 25

DATE ENTERED: 12 November 1984

ENTERED BY: Lt Keith A. Shomper

VERSION: 1.0

Data Item

NAME: inout  
ALIASES: n/a  
DESCRIPTION: A boolean variable denoting the direction of a  
data item (input or output).  
SOURCES: 1.2.3.4  
DESTINATIONS: 1.2.3.4.3  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: boolean  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: level  
ALIASES: n/a  
DESCRIPTION: A integer variable which holds the level value  
when traversing a binary tree structure.  
SOURCES: 1.2.1.3.2  
DESTINATIONS: 1.2.1.2.3, 1.2.1.3.2  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: integer  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: low names  
ALIASES: n/a  
DESCRIPTION: An array or list of child names of a parent process.  
SOURCES: 1.1.1.2.2  
DESTINATIONS: 1.1.2.4  
COMPOSITION: array of names  
PART OF: n/a  
DATA CHARACTERISTIC: array of character string, length 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: menu  
ALIASES: n/a  
DESCRIPTION: The menu position pointer. Keeps track of the user's level and position in the menu structure.  
SOURCES: A21  
DESTINATIONS: A22, A261  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: pointer variable  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: mis list  
ALIASES: n/a  
DESCRIPTION: A list of process names which have no match  
after the comparison routine.  
SOURCES: 1.2.2.5  
DESTINATIONS: none  
COMPOSITION: list of names  
PART OF: n/a  
DATA CHARACTERISTIC: array of character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: mismatch  
ALIASES: n/a  
DESCRIPTION: Data flag signaling and error in a comparison  
of two names.  
SOURCES: 1.2.2.3  
DESTINATIONS: none  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: boolean  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: name  
ALIASES: n/a  
DESCRIPTION: A generic character string variable to hold a  
process name.  
SOURCES: 1.3.2.2, 2.2.3.2  
DESTINATIONS: 1.2.2.3, 1.2.2.5, 1.2.2.6, 1.2.3.3, 1.2.3.4,  
1.2.3.4.3, 1.1.1.2.2, 2.2.3.3, 2.2.3.4,  
2.4.2.2, 2.4.3.3, 2.4.1.5, 2.5.3, 2.5.3.1,  
1.3.2.2, 4.2.2.3  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: name 1  
ALIASES: n/a  
DESCRIPTION: A generic variable to hold the value of a  
single process name.  
SOURCES: 2.4, 2.4.3  
DESTINATIONS: 2.4.2, 2.4.3, 2.4.4, 2.4.5, 2.4.3.3, 2.4.2.3,  
2.4.2.4  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0



#### Data Item

NAME: name 2  
ALIASES: n/a  
DESCRIPTION: A generic variable to hold the value of a  
single process name, usually used in  
conjunction with name 1.  
SOURCES: 2.4, 2.4.3  
DESTINATIONS: 2.4.2, 2.4.3, 2.4.3.3, 2.4.2.3, 2.4.2.4  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, length 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: node  
ALIASES: n/a  
DESCRIPTION: A record variable which contains a name and  
number field; used to hold any generic process  
and its number.  
SOURCES: 1.2.3.4.4  
DESTINATIONS: 1.1.1.2.3, 1.1.2.5, 1.2.3.4.2  
COMPOSITION: name and number  
PART OF: n/a  
DATA CHARACTERISTIC: record variable with name and number  
field.  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: node 1  
ALIASES: n/a  
DESCRIPTION: A generic record variable to hold the value of  
a process name and number.  
SOURCES: 2.4.2  
DESTINATIONS: 2.4.2.2, 2.4.2.3, 2.4.2.4  
COMPOSITION: name and number  
PART OF: n/a  
DATA CHARACTERISTIC: record of character strings  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: node 2  
ALIASES: n/a  
DESCRIPTION: A generic record variable to hold the value of  
a process name and number usually used in  
cunjunction with node 1.  
SOURCES: 2.4.2  
DESTINATIONS: 2.4.2.2, 2.4.3.2, 2.4.2.4, 2.4.2.5  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: record of character strings  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: node list  
ALIASES: n/a  
DESCRIPTION: A list of process names and numbers .  
SOURCES: 1.2.3.3, 2.2.3.3, 4.2.2.3  
DESTINATIONS: 1.2.3.4, 1.2.3.4.1, 1.2.3.4.4, 2.2.3.4,  
4.2.2.4, 2.4.4.5  
COMPOSITION: names and numbers  
PART OF: n/a  
DATA CHARACTERISTIC: array of records of character strings  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Data Item

NAME: parent  
ALIASES: n/a  
DESCRIPTION: A name variable to hold a parent process name.  
SOURCES: 2.4.1, 2.5.1  
DESTINATIONS: 2.4.4.2  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: pair list  
ALIASES: n/a  
DESCRIPTION: A pair of process names from different  
                  lifecycle phases, used in the comparison for  
                  functional completeness across phase boundaries.  
SOURCES: 2.5.2  
DESTINATIONS: none  
COMPOSITION: a pair of process names  
PART OF: n/a  
DATA CHARACTERISTIC: array of character strings  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: process list  
ALIASES: n/a  
DESCRIPTION: An array of process names, usually sibling  
                  processes.  
SOURCES: 1.2.2.2  
DESTINATIONS: 1.3.2.2  
COMPOSITION: array of names  
PART OF: n/a  
DATA CHARACTERISTIC: array of character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: project name  
ALIASES: n/a  
DESCRIPTION: The name which the Test Methodology uses to find information on a particular project.  
SOURCES: A1  
DESTINATIONS: A2, A3, A26, A262, 1.1.1, 1.1.1.2, 1.1.1.2.1, 1.1.2, 1.1.3, 1.1.4, 1.1.4.2, 1.2.1.2, 1.2.2, 1.2.2.2, 1.2.3, 1.2.3.2, 1.2.4, 1.2.4.1, 1.2.4.2, 1.2.4.3, 1.2.4.4, 1.3.1, 1.3.2, 2.2.3, 2.2.3.2, 2.4, 2.4.2, 2.4.3, 2.4.4, 2.4.4.1, 2.5, 2.5.2, 2.5.4, 2.5.4.1, and 4.2.2  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 12  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME: src  
ALIASES: n/a  
DESCRIPTION: A process name variable which denotes the source of a data item.  
SOURCES: 1.2.3.4, 1.2.3.4.3, 2.2.3.4  
DESTINATIONS: none  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC: character string, lenght 25  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

Data Item

NAME:tree root

ALIASES: n/a

DESCRIPTION: A pointer variable to the head of a tree data structure.

SOURCES: 1.1.1.2, 1.1.1.2.3, 1.2.1.2, 1.2.1.2.2, 1.2.1.2.3

DESTINATIONS: 1.1.1.3, 1.1.1.2.3, 1.1.1.3.1, 1.1.3.3,  
1.2.1.2, 1.2.1.2.2, 1.2.1.2.3, 1.2.1.3.2

COMPOSITION: n/a

PART OF: n/a

DATA CHARACTERISTIC: pointer variable

DATE ENTERED: 12 November 1984

ENTERED BY: Lt Keith A. Shomper

VERSION: 1.0

Data Item

NAME:  
ALIASES: n/a  
DESCRIPTION:  
SOURCES:  
DESTINATIONS:  
COMPOSITION: n/a  
PART OF: n/a  
DATA CHARACTERISTIC:  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: add act verb  
NUMBER: 1.2.2.4  
DESCRIPTION: Allows the user to interactively supplement the  
active verb file if a known active verb is  
missing.  
INPUTS: name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: active verb file  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: add mis list  
NUMBER: 1.2.2.5  
DESCRIPTION: When a process name does not pass the active  
verb test, that name is placed in a miss file  
by this routine.  
INPUTS: name  
OUTPUTS: mis list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



#### Process Item

NAME: build bf tree  
NUMBER: 1.1.1.2.3  
DESCRIPTION: Creates a binary tree for the storage and  
                    sorting of a model's decomposition.  
INPUTS: tree root, node  
OUTPUTS: tree root  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: build level  
NUMBER: 1.2.1.2.3  
DESCRIPTION: The recursive routine which builds each level  
                    of the leveling model tree.  
INPUTS: tree root, level  
OUTPUTS: tree root  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1.2, 1.2.1.2.3  
PROCESSES CALLED: 1.2.1.2.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: build menu  
NUMBER: A21  
DESCRIPTION: The top-most routine in the recursive algorithm  
to build the Methodology's menu data structure.  
INPUTS: menu  
OUTPUTS: menu  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: build root  
NUMBER: 1.2.1.2.2  
DESCRIPTION: The executive in the interactive process of  
building the model tree.  
INPUTS: tree root, head node  
OUTPUTS: tree root  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

AD-A152 857

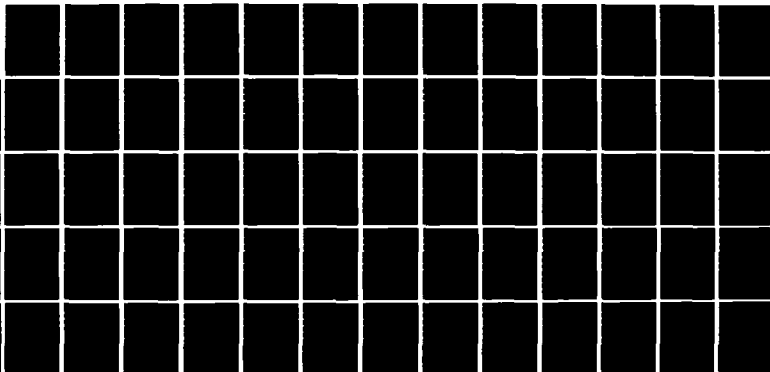
A TEST METHODOLOGY FOR AN AUTOMATED AND INTERACTIVE  
SOFTWARE DEVELOPMENT ENVIRONMENT(U) AIR FORCE INST OF  
TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI  
K A SHOMPER DEC 84 AFIT/GCS/ENG/84D-26

4/4

UNCLASSIFIED

F/G 9/2

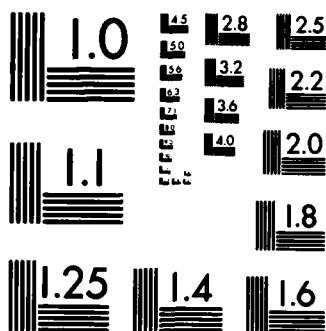
NL



END

FORM

071



MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

#### Process Item

NAME: bound test  
NUMBER: 4.3.1  
DESCRIPTION: A tutorial on boundry value analysis.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: build trace  
NUMBER: 2.4.5  
DESCRIPTION: The routine to build traceability matrix.  
INPUTS: name 1  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: build tree  
NUMBER: 1.2.1.2  
DESCRIPTION: Builds the models leveling tree for to be  
displayed to the user.  
INPUTS: project name, tree root  
OUTPUTS: tree root  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1  
PROCESSES CALLED: 1.1.1.2.1, 1.2.1.2.2., and 1.2.1.2.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: change name  
NUMBER: 1.2.2.6  
DESCRIPTION: Allows the user to interactively change the  
name of a process to an active verb.  
INPUTS: name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: closing message  
NUMBER: A3  
DESCRIPTION: Gives a concluding message to the Test  
Methodology user and informs him where the  
diagnostic output is stored.  
INPUTS: project message  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A0  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: cnt pro  
NUMBER: 1.3.1  
DESCRIPTION: Test for a specified number of processes per  
diagram.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.2.2, 1.1.1.2.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: cnt pro  
NUMBER: 1.3.1  
DESCRIPTION: Test for a specified number of processes per  
                    diagram.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.2.2, 1.1.1.2.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: compare entries  
NUMBER: 2.4.2.2  
DESCRIPTION: A comparison routine for matching process  
                    names.  
INPUTS: node 1, node 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



#### Process Item

NAME: compare process name  
NUMBER: 1.2.2.3  
DESCRIPTION: A comparison of a process name with the active verb file to verify if the process name is indeed an active verb.  
INPUTS: name  
OUTPUTS: mismatch  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: comp int  
NUMBER: 2.5.4  
DESCRIPTION: The user intensive routine to compare a process's functional intent.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: complete  
NUMBER: 2.5  
DESCRIPTION: The executive routine for the completeness tests.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 2.5.1, 2.5.2, 2.5.3, and 2.5.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: DD main module  
NUMBER: D0  
DESCRIPTION: The executive module for the Data Dictionary Generation Tool.  
INPUTS: consistency errors  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: det asso for unmtch  
NUMBER: 2.4.4  
DESCRIPTION: A user routine to find obscure matches.  
INPUTS: name 1  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4  
PROCESSES CALLED: 2.4.4.1, 2.4.4.2, and 2.4.4.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: det src dst  
NUMBER: 1.2.3.4  
DESCRIPTION: Finds the source and the destination of each of  
the data items in the model.  
INPUTS: name, node list  
OUTPUTS: src, dst  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3  
PROCESSES CALLED: 1.2.3.4.1, 1.2.3.4.2, 1.2.3.4.3, and  
1.2.3.4.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: det src dst (SC)  
NUMBER: 2.2.3.4  
DESCRIPTION: Determines the source and destinations of data items.  
INPUTS: name, node list  
OUTPUTS: src, dst  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: display options  
NUMBER: A22  
DESCRIPTION: Writes the current list of options in the menu to the user's terminal screen.  
INPUTS: menu and command buffer  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: display tree  
NUMBER: 1.1.1.3.1  
DESCRIPTION: Causes the binary tree created in build bf tree  
to be written to the user's diagnostic file.  
INPUTS: tree root  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: edt editor  
NUMBER: 4.4  
DESCRIPTION: The EDT editor  
INPUTS: file name  
OUTPUTS: file name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: error test  
NUMBER: 4.3.3  
DESCRIPTION: A tutorial on error recovery testing.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: examine DFD  
NUMBER: 1.2.4.3  
DESCRIPTION: Provide tutorial information for the  
                  examination of the DFD diagram.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: examine SADT  
NUMBER: 1.2.4.2  
DESCRIPTION: Provides tutorial information for the  
                    examination of the SADT diagram.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: examine SC  
NUMBER: 1.2.4.4  
DESCRIPTION: Provides tutorial information for the  
                    examination of the SC diagram.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: exam level  
NUMBER: 1.2.1  
DESCRIPTION: Tests for appropriate leveling in the user's  
model decomposition.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.1.1, 1.2.1.2, and 1.2.1.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: execute command  
NUMBER: A26  
DESCRIPTION: Controls either the call to select for  
execution of a component or to move ptr with  
respect to the user's position within the menu  
structure.  
INPUTS: project name, menu, command buffer, command  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: move ptr, select  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



### Process Item

NAME: find interface boundaries  
NUMBER: 1.1.2.4  
DESCRIPTION: An algorithm to determine the hierarchical  
                  boundry interfaces by using succeeding matches  
                  for ICOMS.  
INPUTS: control parms, low names  
OUTPUTS: ICOMS  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: find last  
NUMBER: 1.2.3.4.4  
DESCRIPTION: Determines the last node in a sata item' path.  
INPUTS: node list  
OUTPUTS: node  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: find low names  
NUMBER: 1.1.1.2.2  
DESCRIPTION: Returns the names and numbers of the decomposition of a higher-level process name.  
INPUTS: head node  
OUTPUTS: low names  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: find nodes  
NUMBER: 1.2.3.3  
DESCRIPTION: Finds the processes associated with a particular data item.  
INPUTS: name  
OUTPUTS: node list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: find nodes (SC)  
NUMBER: 2.2.3.3  
DESCRIPTION: The SC version to find the nodes that are  
                  associated with a particular data item.  
INPUTS: name  
OUTPUTS: node list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: get head node  
NUMBER: 1.1.1.2.1  
DESCRIPTION: Retrieves the top-most process in the  
                  hierarchy.  
INPUTS: project name  
OUTPUTS: head node  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: get process data  
NUMBER: 1.3.2.2  
DESCRIPTION: Retrieves the data items associated with a list  
of processes.  
INPUTS: process list  
OUTPUTS: name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.3.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: get process list  
NUMBER: 1.2.2.2  
DESCRIPTION: Retrieves the list of process names for a given  
model.  
INPUTS: project name  
OUTPUTS: process list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: get pro pairs  
NUMBER: 2.5.2  
DESCRIPTION: Joins together model names from two phases for  
a comparison of functional intent.  
INPUTS: project name  
OUTPUTS: pair list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: int cons intro  
NUMBER: 1.1.3.1  
DESCRIPTION: Introduces the test for internal consistency by  
writing the appropriate template to the user's  
vidio terminal.  
INPUTS: none  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: intro complete  
NUMBER: 2.5.1  
DESCRIPTION: Introduces complete with the appropriate  
template displayed to the user's screen.  
INPUTS: none  
OUTPUTS: parent  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: introduce data dictionary  
NUMBER: 1.1.4.1  
DESCRIPTION: INPUTS: Introduces the DD Tool by provide  
tutorial information.  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: introduce test txt for err  
NUMBER: 1.2.4.1  
DESCRIPTION: Introduces test txt err with the appropriate  
template.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: Introduction to Test Methodology  
NUMBER: A1  
DESCRIPTION: Provides the opening template to the  
Methodology and queries the user for the  
project name.  
INPUTS:  
OUTPUTS: project name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A0  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: intro el  
NUMBER: 1.2.1.1  
DESCRIPTION: Introduces exam level with the appropriate  
template.  
INPUTS: none  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: intro int test  
NUMBER: 4.2.2.1  
DESCRIPTION: Introduces int test with the appropriatly  
displayed template.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 4.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



Process Item

NAME: intro print tree  
NUMBER: 1.2.1.3.1  
DESCRIPTION: Introduces print tree with the display of the appropriate template to the user's screen.  
INPUTS: none  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: intro rec data  
NUMBER: 1.2.3.1  
DESCRIPTION: Introduces rec data flow with the appropriate template.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: intro rec data  
NUMBER: 2.2.3.1  
DESCRIPTION: Introduces rec data flow, the SC version with  
the appropriate template.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: intro trace  
NUMBER: 2.4.1  
DESCRIPTION: Introduce trace by displaying the appropriate  
template.  
INPUTS: none  
OUTPUTS: parent  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: int test  
NUMBER: 4.2.2  
DESCRIPTION: The executive of the interface testing routine.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 4.2.2.1, 1.2.2.2, 4.2.2.3, 4.2.2.4, and  
2.4.4.5  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: main  
NUMBER: A0  
DESCRIPTION: This module controls the executive level of the  
Test Methodology; it is the top-most module.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: none  
PROCESSES CALLED: A1, A2, and A3  
FILES READ: menu and helpfile  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: menu help  
NUMBER: A25  
DESCRIPTION: Provides the help facility for the menu options.  
INPUTS: command buffer  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: move ptr  
NUMBER: A261  
DESCRIPTION: Moves the menu pointer to the appropriate place in the menu structure to designate the user's position in the menu and to determine his options.  
INPUTS: menu, command buffer, command  
OUTPUTS: command buffer, command  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A26  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: mtch dic pro  
NUMBER: 2.4.2  
DESCRIPTION: The routine to match a model's process names  
with the data dictionary entries.  
INPUTS: name 1, name 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4  
PROCESSES CALLED: 1.1.1.2.1, 2.4.2.2, 2.4.2.3, 2.4.2.4,  
2.4.2.5, and 1.1.1.2.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: mtch dic pro  
NUMBER: 2.4.2  
DESCRIPTION: The routine to match a model's process names  
with the data dictionary entries.  
INPUTS: name 1, name 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4  
PROCESSES CALLED: 1.1.1.2.1, 2.4.2.2, 2.4.2.3, 2.4.2.4,  
2.4.2.5, and 1.1.1.2.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: offer menu  
NUMBER: A2  
DESCRIPTION: This module is the executive for all the the  
                  interactive menu routines.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A0  
PROCESSES CALLED: A21, A22, A23, A24, A25, and A26  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: path test  
NUMBER: 4.3.2  
DESCRIPTION: A tutorial on path testing  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: present call  
NUMBER: 4.2.2.3  
DESCRIPTION: Displays the calling order of the interface  
parameters.  
INPUTS: name  
OUTPUTS: node list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 4.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: present data information  
NUMBER: 2.4.4.1  
DESCRIPTION: Provides data dictionary information about  
selected data items.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: present diagram  
NUMBER: 1.1.3.5  
DESCRIPTION: Display the given diagram and provides for user  
interaction with that diagram to modify it.  
INPUTS: control parms  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: present order  
NUMBER: 2.4.4.5  
DESCRIPTION: Displays any problems with the interfaces  
calling order.  
INPUTS: node list  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 4.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



#### Process Item

NAME: present parm list  
NUMBER: 4.2.2.4  
DESCRIPTION: Presents the defined calling order in the data dictionary.  
INPUTS: node list  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 4.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: print level  
NUMBER: 1.2.1.3.2  
DESCRIPTION: The recursive routine which prints the leveling model.  
INPUTS: tree root, head node, level,  
OUTPUTS: head node, level  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1.3, 1.2.1.3.2  
PROCESSES CALLED: 1.2.1.3.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: print tree  
NUMBER: 1.2.1.3  
DESCRIPTION: Print the model tree so the user may visually  
examine the leveling.  
INPUTS: tree root  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.1  
PROCESSES CALLED: 1.2.1.3.1 and 1.2.1.3.2  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: provide access to DD  
NUMBER: 1.1.4.2  
DESCRIPTION: Sets up the connection with the DD Tool.  
INPUTS: project name  
OUTPUTS: consistency errors  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: provide consistency information  
NUMBER: 1.1.3.2  
DESCRIPTION: A tutorial of information to aid the user  
indetermining wether a diagram is internally  
consistent.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: provide data access  
NUMBER: 2.4.4.2  
DESCRIPTION: Allows the user to query the data base for a  
greater understanding of the data  
relationships.  
INPUTS: parent  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: provide note taking  
NUMBER: 1.1.1.3.3  
DESCRIPTION: A slightly modified version of write instructions.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: put item  
NUMBER: 1.2.3.4.2  
DESCRIPTION: Puts a data item into the form for trace to find the data's sources and destinations.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: read command  
NUMBER: A23  
DESCRIPTION: Reads the user's command from the keyboard.  
INPUTS: none  
OUTPUTS: command  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: rec data flow  
NUMBER: 1.2.3  
DESCRIPTION: Tests for conservation of data.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.3.1, 1.2.3.2, 1.2.3.3, and 1.2.3.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: rec data flow (SC)  
NUMBER: 2.2.3  
DESCRIPTION: The structure chart version of rec data flow.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 2.2.3.1, 2.2.3.2, 2.2.3.3, and 2.2.3.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: record aliases  
NUMBER: 1.1.4  
DESCRIPTION: Prepares the user to enter into the Data  
Dictionary Generation Tool from the  
Methodology.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.1.4.1, 1.1.4.2, and D0  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: record matched  
NUMBER: 2.4.2.3  
DESCRIPTION: Records the matched process names.  
INPUTS: node 1, node 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: record similar  
NUMBER: 2.4.2.4  
DESCRIPTION: Records those process names which are somewhat  
                  alike, but not exactly.  
INPUTS: node 1, node 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: record unmatched  
NUMBER: 2.4.2.5  
DESCRIPTION: Records those process names which do not match  
                    at all.  
INPUTS: node 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: ret func info  
NUMBER: 2.5.3  
DESCRIPTION: Retrieves the functional information associated  
                    with the pair of process names.  
INPUTS: name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5  
PROCESSES CALLED: 2.5.3.1, 1.1.1.2.2, 1.3.2.2, and 2.5.3.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



#### Process Item

NAME: retrieve data items  
NUMBER: 1.2.3.2  
DESCRIPTION: Retrieves one-by-one each of the data items.  
INPUTS: project name  
OUTPUTS: data list  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: retrieve data items  
NUMBER: 2.2.3.2  
DESCRIPTION: Gets all the data of a given project.  
INPUTS: project name  
OUTPUTS: name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.2.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: select  
NUMBER: A262  
DESCRIPTION: This routines only function is to provive an interface with the Test Methodology's component tests.  
INPUTS: project name, command  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A26  
PROCESSES CALLED:  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: select item  
NUMBER: 1.2.3.4.1  
DESCRIPTION: Retrieve a single node from a list.  
INPUTS: node list  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: select next diagram  
NUMBER: 1.1.3.4  
DESCRIPTION: Brings up the next diagram on the user's  
screen.  
INPUTS: none  
OUTPUTS: name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: simulation test  
NUMBER: 4.3.5  
DESCRIPTION: A tutorial on support for simulation testing.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: take notes  
NUMBER: 2.4.4.3  
DESCRIPTION: An interactive note-taking routine.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: note file  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: test case gen  
NUMBER: 4.3.6  
DESCRIPTION: A test case generator.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: test for interface consistency  
NUMBER: 1.1.2.5  
DESCRIPTION: This routine makes the actual test for consistency once the data has been processed by the preceding routines.  
INPUTS: node, ICOMS  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES:  
PROCESSES CALLED:  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: test txt err  
NUMBER: 1.2.4  
DESCRIPTION: Tests for textual errors in a diagram by employing the user's knowledge and tutorial information.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.4.1, 1.2.4.2, 1.2.4.3, and 1.2.4.4  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: trace  
NUMBER: 2.4  
DESCRIPTION: The tracabilty tests for the Test Methodology.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 2.4.1, 2.4.2, 2.4.3, 2.4.4, and 2.4.5  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: trace input/output  
NUMBER: 1.2.3.4.3  
DESCRIPTION: Finds the sources and the destinations for a particular data item by tracing the data item's path through the model.  
INPUTS: name, inout  
OUTPUTS: src, dst  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.3.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: traverse tree  
NUMBER: 1.1.3.3  
DESCRIPTION: Steps the user through each diagram in the decomposition order.  
INPUTS: tree root  
OUTPUTS: name  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: trim  
NUMBER: A24  
DESCRIPTION: Trims the six-character command word down to a unique three-character command word to save on storage.  
INPUTS: command  
OUTPUTS: command  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver act nam  
NUMBER: 1.2.2  
DESCRIPTION: A test to verify that active names are being  
used in the model's process names.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.2.2.1, 1.2.2.2, 1.2.2.3, 1.2.2.4,  
1.2.2.5, and 1.2.2.6  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver act nam intro  
NUMBER: 1.2.2.1  
DESCRIPTION: Introduces ver act name by displaying a  
template to the user's screen.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.2.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



### Process Item

NAME: ver dat name  
NUMBER: 2.4.3.3  
DESCRIPTION: A comparison routine for the model's data  
names.  
INPUTS: name, name 1, name 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.4.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver int  
NUMBER: 1.1.2  
DESCRIPTION: The component test to verify the model's  
interfaces along the hierarchical lines.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.1.2.1, 1.1.1.2.1, 1.1.1.2.2, 1.1.2.4,  
1.1.2.5  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: ver int cons  
NUMBER: 1.1.3  
DESCRIPTION: A test for internal consistency. This test  
relies mostly on user interaction and  
perception.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.1.3.1, 1.1.3.2, 1.1.3.3, 1.1.3.4, and  
1.1.3.5  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: ver int intro  
NUMBER: 1.1.2.1  
DESCRIPTION: Introduces ver int with a template written to  
the user's terminal screen.  
INPUTS: none  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.2  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver name  
NUMBER: 1.1.1.2  
DESCRIPTION: The part of ver nam num that tests on the  
process names.  
INPUTS: project name, control parms  
OUTPUTS: tree root  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1  
PROCESSES CALLED: 1.1.1.2.1, 1.1.1.2.2, and 1.1.1.2.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver nam num  
NUMBER: 1.1.1  
DESCRIPTION: A test to verify the process names and numbers  
for data dictionary entries.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: 1.1.1.1, 1.1.1.2, and 1.1.1.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: Ver nam num intro  
NUMBER: 1.1.1.1  
DESCRIPTION: Introduces ver name num with a template on the user's terminal screen, and allows the user to set control parameters which determine the succeeding actions in this test.  
INPUTS: none  
OUTPUTS: control parms  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

### Process Item

NAME: ver num  
NUMBER: 1.1.1.3  
DESCRIPTION: The part of ver nam num that tests on the process numbers.  
INPUTS: control parms, tree root  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1  
PROCESSES CALLED: 1.1.1.3.1, 1.1.1.3.2, and 1.1.1.3.3  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS BASIC compiler  
NUMBER: 4.2.3.4  
DESCRIPTION: The BASIC compiler  
INPUTS: file name  
OUTPUTS: object file  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS COBAL compiler  
NUMBER: 4.2.3.2  
DESCRIPTION: The COBOL compiler  
INPUTS: file name  
OUTPUTS: object file  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS diff facility  
NUMBER: 4.2.4  
DESCRIPTION: The difference facility  
INPUTS: file name 1, file name 2  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS FORTRAN compiler  
NUMBER: 4.2.3.1  
DESCRIPTION: The FORTRAN compiler  
INPUTS: file name  
OUTPUTS: object file  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS linker  
NUMBER: 4.3.7  
DESCRIPTION: The linker facility.  
INPUTS: file name(s)  
OUTPUTS: executable image  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: VMS PASCAL compiler  
NUMBER: 4.2.3.3  
DESCRIPTION: The PASCAL compiler  
INPUTS: file name  
OUTPUTS: object file  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: A262  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: write instructions  
NUMBER: 1.1.1.3.2  
DESCRIPTION: Allows the user the opportunity to write  
comments to his output file interactively with  
the Methodology.  
INPUTS: none  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 1.1.1.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

#### Process Item

NAME: write lists  
NUMBER: 2.5.3.4  
DESCRIPTION: Write the processes decomposition list to the  
user's screen.  
INPUTS: data list  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0



Process Item

NAME: write message  
NUMBER: 2.5.4.1  
DESCRIPTION: A interactive routine to write messages  
directly to the data base file to be stored  
with the model description.  
INPUTS: project name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5.4  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: note file  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME: write process info  
NUMBER: 2.5.3.1  
DESCRIPTION: Writes the process description information to  
the user's screen.  
INPUTS: name  
OUTPUTS: none  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES: 2.5.3  
PROCESSES CALLED: none  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Process Item

NAME:  
NUMBER:  
DESCRIPTION: INPUTS:  
OUTPUTS:  
GLOBAL DATA USED: NONE  
GLOBAL DATA MODIFIED: NONE  
ALGORITHM: See Appendix C.  
CALLING PROCESSES:  
PROCESSES CALLED:  
FILES READ: NONE  
FILES WRITTEN: NONE  
DATE ENTERED: 12 November 1984  
ENTERED BY: Lt. Keith A. Shomper  
VERSION: 1.0

Appendix E

Configuration Model Justification

## Appendix E

### SDW Configuration Model Description and Justification

The SDW Configuration Model is provided as a framework for the SDW software components and as a guide for the implementation of the SDW. At the top level of the model is the SDW Executive. The SDW Executive is a software component and provides the interface between the SDW user and the SDW itself. The SDW Executive manages and controls all of the other SDW components.

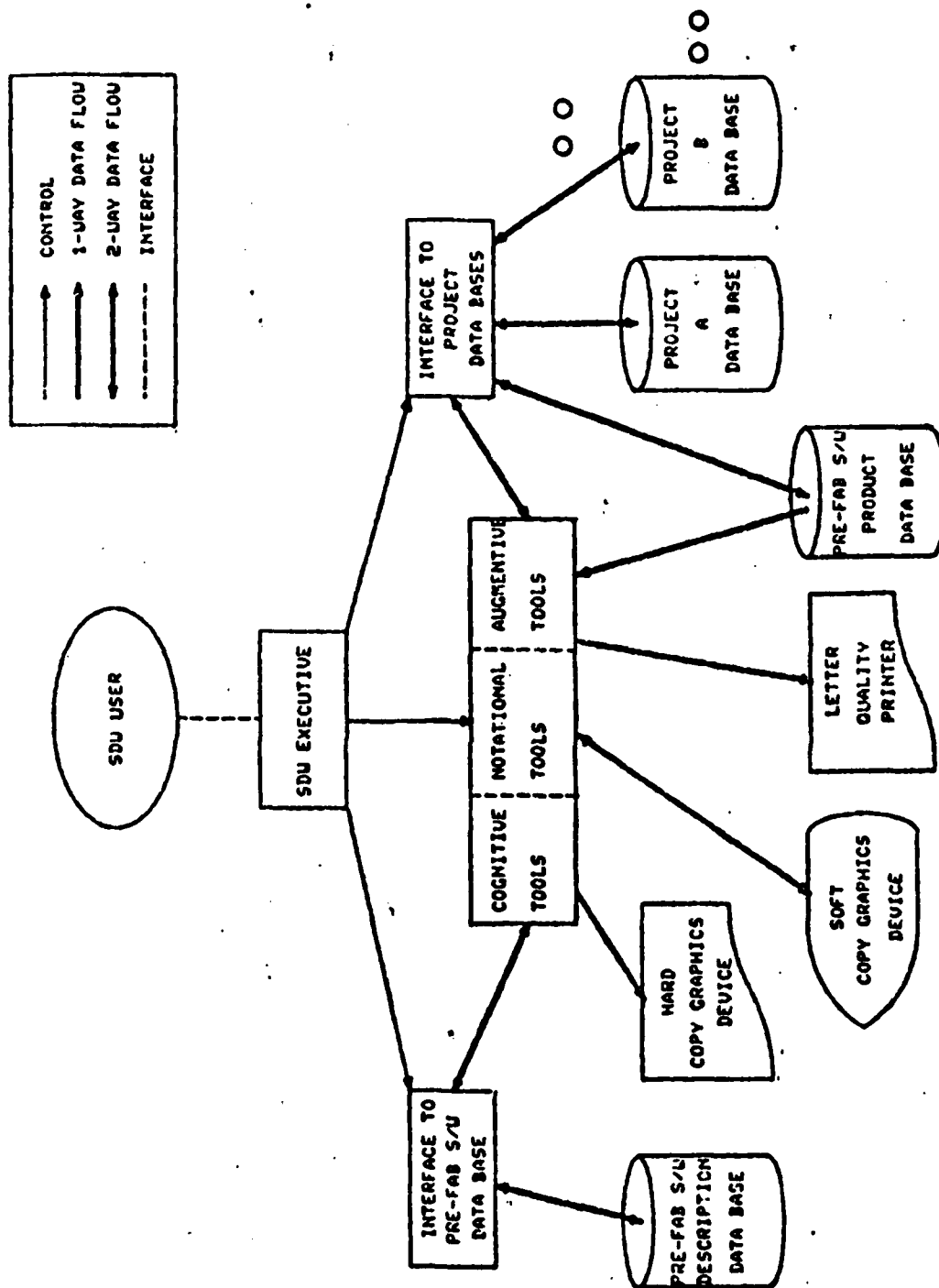
The other software components of the SDW are the Interface to the Pre-Fab Software Description Data Base, the Interface to the Project Data Bases, and the actual automated and interactive tools. The Interface to the Pre\_Fab Software Description Data Base allows the SDW user to search the Pre-Fab Software Description Data Base for descriptions and locations of already written software modules and programs that he may require.

The Interface to the Project Data Base allows both the SDW user and the SDW tools to access the Project Data Bases where all of the software development data and documentation are stored. The Interface to the Project Data Bases also provides for a software transfer link between the Project Data Bases and the Pre-fab Software Product Data Base, where the already constructed software modules and programs are

stored.

The final software component of the SDW is the actual tool set. The individual tools within this tool set are classified by function into three categories. These categories are cognitive tools, notational tools, and augmentive tools (67). The cognitive tools extend the intellectual capabilities of the SDW user by providing automated and interactive facilities to support software engineering methods and techniques. The notational tools assist the SDW user in developing, producing, updating, and maintaining development information. The augmentive tools utilize the computational speed of the computer to check the consistency, precision, and completeness of the development products with great rigor. The SDW tool set is capable of selectively interfacing to other SDW components through data paths. Some tools within the SDW tool set interact with the Interface to the Pre-Fab Software Description Data Base. Most all of the tools communicate with the Interface to the Project Data Bases. Some tools receive software products directly from the Pre-Fab Software Product Data Base. The notational and cognitive tools especially use the three input/output (I/O) hardware devices (the hard copy graphics device, the soft copy graphics device, and the letter quality printers).

The remaining components of the SDW Configuration Model



includes three distinct types of data bases and four hardware I/O devices. The first type of data base is the Pre-Fab Software Description Data Bases. This data base holds the descriptions and locations of existing software packages. This data base assists the SDW user in locating existing software packages that are similar to or may solve part of his particular development project. The Pre-Fab Software Product Data Base is where the actual software packages, described by the Pre-Fab Software Description Data Base, are stored. The last type of data base component in the SDW Configuration Model is the Project Data Base. There exists a separate Project Data Base for each development effort being supported by the SDW. These Project Data Bases hold all of the development documentation and data for the developments beint supported by the SDW.

The hardware components of the SDW Configuration Model are all I/O devices. There are four of these hardware I/O devices. They are a hard copy graphics device, a soft copy graphics device, a leter quality printer, and a standard interactive video terminal. The hard copy graphics device is required for the production of on paper copies of the graphical illustrations of the development efforts supported by the SDW. The soft copy graphics device is required for the displaying and editing of these graphical illustrations on a video dislplay. the letter quality

printer is used for producing copies of software development documentation. The high level of quality is required because this documentation must be included in formal manuscripts such as theses. The last hardware I/O device that is identified by the the SDW Configuration Model is the standard interactive video terminal, realized in the model as the component labeled User. This component is referred to as User because it is the usual device used to interface with the SDW. Of course, separate terminals are required for each concurrent User of the SDW.

The SDW Configuration Model provides a Framework for the developing of the software and data base components of the SDW. These are the components that are the emphasis of this initial development effort of the SDW. However, the model also specifies the required SDW system structure that must be present to satisfy the objectives and concerns of the SDW development effort as stated in Chapter (25). The next section of this chapter (Chapter 3) explains how each of these objectives and concerns are addressed by the SDW design. References are made in that section to the SDW Configuration Model and how its components are used to satisfy these objectives and concerns.

This appendix originally appeared as textual justification for the Configuration Model in (25).



## Bibliography

1. Abbott, Russell J. "Program Design by Informal English Descriptions," Communications of the ACM, Vol. 26, No. 11: 882-894 (November 1983).
2. Adrion, W. Richards, et al. "Validation, Verification, and Testing of Computer Software," ACM Computing Surveys, Vol. 14, No. 2: 159-192 (June 1982).
3. Alberts, David S. "The Economics of Software Quality Assurance," Tutorial: Software Testing and Validation, Second Edition, edited by Edward Miller and William E. Howden. 415-424. IEEE Press New York NY, 1981.
4. ANSI/IEEE Standard. IEEE Standard Glossary of Software Engineering Terminology, Std. 729-1983. IEEE Press New York NY, February 1983.
5. Bergland, G. D. "A Guided Tour of Program Design Methodologies," Computer, Vol. 14, No. 10: 13-37 (October 1981).
6. Biewald, J, et al. "Application of the Specification and Design Techniques EPOS to a Process Control Problem," Proceedings IFAC Symposium Digital Applications to Process Control. 517-522. Pergamon Press, Oxford-New York, 1980.
7. Boehm, Barry W. "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12: 1226-1241 (December 1976).
8. Caine, S. H, E. K. Gordon. "PDL - A Tool for Software Design," Proceedings, National Computer Conference, Vol. 44. 271-276. AFIPS Press 1975.
9. Chapin, N. "Flowcharting with the ANSI Standard: A Tutorial," ACM Computing Surveys, Vol. 2, No. 2: 119-146 (June 1970).
10. Chapin, N. "Function Parsing in Structured Design," Infotech State of the Art Report on Structured Analysis and Design, Vol. 2: 25-43 (Infotech Information Ltd: Berkshire, England, 1978).
11. Culik, K. "On Formal and Informal Proofs for Program Correctness," Sigplan Notices, Vol. 18, No. 1: 23-27 (January 1983).

12. Davis, Alan M. "The Design of a Family of Application-Oriented Requirements Languages," Computer, Vol. 15, No. 5: 21-28 (May 1982).
13. Davis, Richard M. Thesis Projects in Science and Engineering. New York: St Martin's Press, 1980.
14. Dale, Nell, David Orshalick. Introduction to Pascal and Structured Design. Lexington, Mass: D.C. Heath and Co. 1983.
15. Degano, P, C. Levi. "Software Development and Testing in an Integrated Programming Environment," Computer Program Testing, Proceedings of the Summer School on Computer Program Testing. 251-263 Amsterdam: North Holland, 1981.
16. DeWolf, J. Barton. "Requirements Specification and Preliminary Design for Real-Time Systems," IEEE Comp Soc 177. 17-23. IEEE Press New York NY, 1977.
17. Department of the Air Force (AU), Air Force Institute of Technology. Style Guide for Theses and Dissertations. Wright-Patterson AFB OH, 1983.
18. Deutsch, Michael S. Software Verification and Validation edited by Randall W. Jensen. Englewood Cliffs NJ: Prentice Hall, Inc. 1982.
19. Duncan A. G, J. S. Hutchison. "Using Attributed Grammars to Test Designs and Implementation," 5th International Conference on Software Engineering. 170-178. IEEE Press New York NY, 1981.
20. Fairley, Richard E. "Software Testing Tools," Computer Program Testing, Proceedings of the Summer School on Computer Program Testing. 151-186. Amsterdam: North Holland 1981.
21. Foley, J. D, A. VanDam. Fundamentals of Interactive Computer Graphics. Reading, Mass: Addison-Wesley, 1983.
22. Futamura, Y. et al. "Development of Computer Programs by PAD (Problem Analysis Diagram)," Proceedings of the Fifth International Software Engineering Conference. 325-332. IEEE Press New York, NY, 1981.

23. Gilb, T. "Design by Objective: A Structured Systems Architecture Approach," Infotech State of the Art Reports: (Infotech Information Ltd: Berkshire, England, 1978).
24. Gomma, Hassan, B. H. Scott. "Prototyping as a Tool in the Specification of User Requirements," Fifth International Conference on Software Engineering. 333-339. IEEE Press New York NY, 1981.
25. Hadfield, Lt Steven M. An Interactive and Automated Software Development Environment, MS Thesis GCS/EE/82D-17. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982
26. Hamilton, M, S. Zeldin. "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol SE-2, No 1: 9-31 (March 1976).
27. Hamlet, Dick. "Debugging 'Level' : Step-wise Debugging," ACM Software Engineering Notes, Vol. 8, No. 4: 4-8 (August 1983).
28. Hecht, Herbert. "Software Testing and Documentation, Software Test Documentation: Standards Will Help," Computer, Vol. 12, No. 13: 98-107 (March 1979).
29. "HIPO - A Design Aid and Documation Technique," IBM Report GC20-1851-0, October 1974.
30. Howden, William E. "Introduction to Software Validation." Tutorial: Software Testing and Validation Techniques, Second Edition. 1-2. IEEE Press New York NY, 1981.
31. Howden, William E. "Lifecycle Software Validation," Computer, Vol 15: No 2: 71-78 (February 1982).
32. Horowitz, Ellis, Sartaj Sahni. Fundamentals of Data Structures. Rockville, MA: Computer Science Pres, Inc, 1983.
33. Howley, Paul P. "A Comprehensive Software Testing Methodology," Second Software Engineering Standards Workshop. 156-163. IEEE Press New York NY, 1983.
34. Huang, J. C. "Program Instrumentation and Software Testing," Computer, Vol. 11, No. 4: 25-32 (April 1978).

35. Jensen, Randolph W. "Structured Programming," Computer, Vol. 14, No. 3: 31-48 (March 1981).
36. Kernighan, Brian W, P. J. Plauger. Elements of Programming Style, Second Edition. New York: McGraw Hill, 1978.
37. Kernighan, Brian W, P. J. Plauger. Software Tools. Reading, Mass: Addison-Wesley, 1976.
38. Krygiel, Annette J. Lessons Learned on the Road to a Modern Programming Environment. AD-A111102, January 1982.
39. Lauber, Rudolf J. "Development Support Systems," Computer, Vol 15, No 5: 36-46 (May 1982).
40. Ledgard, H. F. "The Case of Structured Programming," BIT, Vol 13: 45-47 (1973).
41. Lehman, John H. "How Software Projects are Really Managed," Datamation, Vol. 25, No. 1: 119-129 (January 1979).
42. McCracken, Daniel D. A Guide to FORTRAN IV Programming. New York, NY: Kohn Wiley and Sons, Inc, 1965.
43. Millard, David P. "Automated Documentation for Real-Time Software," IEEE Southeastcon '83. 78-80. IEEE Press New York NY, 1983.
44. Moore, Paul. Extension of the Software Development Workbench to Include MicroComputer Workstations, MS Thesis GCS/ENG/84D-XX. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1982.
45. Morgan, D. E, D. J. Taylor. "A Survey of Methods of Achieving Reliable Software," Computer, Vol. 10, No. 2: 44-53 (February 1977).
46. Parisi, Vincent M. Development of a Computer Aided Design Package for Control System Design and Analysis for Use on a Personal Computer, MS Thesis GE/EE/83D-53. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. December 1983.

47. Parnas, D. L. "On Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 5, No. 12: 1053-1058 (December 1972).
48. Peters, Lawrence J. Software Design: Methods and Techniques. New York NY: Yourdon Press, 1981.
49. Pressman, Roger S. Software Engineering a Practitioners Approach. McGraw Hill, 1982.
50. Prudhomme, Robert R. "Software Verification and Validation and SQA," American Society for Quality Control 34 Annual Technical Conference Transactions. 397-404. American Society Quality Control, Inc. Milwaukee WI, 1980.
51. Ross, D. T. "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. 3, No. 1. 36-54. IEEE Press New York NY, 1977.
52. Ross, D, K. Schoman. "Structured Analysis for Requirements Definition," IEEE Transactions on Software Engineering, Vol. 3, No. 1. 6-15. IEEE Press New York NY, 1977.
53. Rubey, Raymond J. Software Acquisition Guide for Managers, Softech Inc, Fairborn OH, May 1984.
54. Scharer, Laura. "Pinpointing Requirements," Datamation, Vol. 27, No. 4: 139-151 (April 1981).
55. Silverthorn, Lee. "Improving Software Project Management by Emphasizing Testing Procedures," Second Annual Phoenix Conference on Computers and Communications Conference Proceedings. 149-153. IEEE Press New York NY, 1983.
56. Smith, Mark K. et al. "An Approach to Transfer Verification and Validation Technology," AFIPS Conference Proceedings, Vol. 50. 367-373. Arlington VA, 1981.
57. Teichroew, D. and A. Hershey. "PSL/PSA, A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1: 41-48. (January 1977).

58. Thomas, Wayne. An Automated Interactive Software Engineering Tool To Generate Data Dictionaries, MS Thesis GSC/ENG/84D-XX. School of Engineering, Air Force Institute of Tech Wright-Patterson AFB, OH December 1984
59. Van Lansweerde, Axel. "Automating the Production of Application Software: Some Insights Part 3," Technology and Science, Vol. 2, No. 2: 77-88. North Oxford Academic, 1983.
60. Weinberg, Victor. Structured Analysis. New York: Yourdon Press, 1979.
61. Wile, David S. "Program Developments: Formal Explanations of Implementations," Communications of the ACM, Vol. 26, No. 11: 902-911 (November 1983).
62. Willis, R. R. "AIDES - Computer Aided Design of Software Systems," Proceeding of the Symposium on Software Engineering Environments. Elsevier-North Holland, 1980.
63. Wirth, N. "On the Composition of Well-Structured Programs," Computing Surveys, Vol. 6, No. 4: 247-259 (December 1974).
64. Wilson, Robert E. Continued Development of an Interactive Control Engineering Computer Analysis Package (ICECAP) for Discrete and Continuous Systems, MS Thesis GE/EE/83D-72. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. December 1983.
65. Yourdon, Edward, Larry L. Constantine. Structure Design, Second Edition. New York NY: Yourdon Press, 1978.
66. Yourdon, Edward. Techniques of Program Structure and Design. Englewood Cliffs, NJ: Prentice-Hall, Inc, 1975.
67. Yushchenko, E. L, I. V. Kasatkina. "Current Methods for Proving Program Correctness," Cybernetics, Vol. 16, No. 6: 832-859 (December 1980).
68. Zelkowitz, Marvin V. et al. Principles of Software Engineering and Design. Englewood Cliffs NJ: Prentice Hall, Inc. 1979.

## VITA

Lt. Keith A Shomper was born on 4 December 1961 in Halifax, Pennsylvania. He graduated from Columbine High School in June of 1979, and afterwards attended the University of Northern Colorado, where he in June of 1983 earned his Bachelor of Arts degree in Mathematics. Upon graduation he received his commission in the USAF as an ROTC Distinguished Graduate. His first assignment was to the Air Force Institute of Technology, entering in June of 1983.

Permanent Address: 6452 W. Alder Ave.

Littleton, Colorado 80123

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved Public Release; Distribution Unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/84D-26			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering		6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION		
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson, AFB, Ohio 45433			7b. ADDRESS (City, State and ZIP Code)		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER		
8c. ADDRESS (City, State and ZIP Code)			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.
11. TITLE (Include Security Classification) See Box 19					
12. PERSONAL AUTHOR(S) See Box 19					
13a. TYPE OF REPORT MS Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Yr., Mo., Day) 1984 December	
15. PAGE COUNT 353					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB. GR.	Lifecycle Testing, Software Development Environments, Software Validation		
9	2				
19. ABSTRACT (Continue on reverse if necessary and identify by block number)					
<p>Title: A Test Methodology for an Automated and Interactive Software Development Environment</p> <p>Author: Lt. Keith A. Shomper</p> <p>Abstract: The purpose of this investigation is to examine the concept of lifecycle testing, in particular in the AFIT education/research environment. A result of this investigation is the SDW Test Methodology, an automated/interactive testing tool to aid the software engineer in lifecycle testing.</p>					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION		
22a. NAME OF RESPONSIBLE INDIVIDUAL			22b. TELEPHONE NUMBER (Include Area Code)		22c. OFFICE SYMBOL

Approved for public release: 1AW AFR 190-1  
 Lynn E. Wolaver 21 Feb 85  
 Dean for Research and Professional Development  
 Air Force Institute of Technology (AFIT)  
 Wright-Patterson AFB OH 45433



(J9 continued)

Five areas of testing are identified during this investigation which are common to software development in the analysis, design and implementation phases. These areas are consistency, correctness, clearness, completeness, and traceability. By providing automated and interactive functions to test for the errors associated with these areas, the AFIT student software engineer is relieved from the hand-testing techniques which heretofore have been employed. The Tests Methodology supports the latter half of the lifecycle testing by providing interfaces to the Test Methodology for an assortment of static and dynamic analysis testing tools.

On a wider scale, the ideas and conclusions presented herein are applicable in all software development environments employing a structured approach to analysis, design and implementation. However, the software tests are not generally applicable to all environments and may require modifications. The SDW Test Methodology is initially hosted on a VAX VMS <sup>11</sup>11/780 and requires interfaces to the INGRES Relational Database Manager and the Data Dictionary Generation Tool (58).

*page 11/780 included in it*

**END**

**FILMED**

**5-85**

**DTIC**